Compiler Strategies for Transport Triggered Architectures

Johan Janssen

Compiler Strategies for Transport Triggered Architectures

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.F. Wakker,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 17 september 2001 om 16:00 uur

door

Johannes Antonius Andreas Jozef JANSSEN

elektrotechnisch ingenieur
geboren te Wamel

Dit proefschrift is goedgekeurd door de promotoren:
Prof.dr.ir. A.J. van de Goor
Prof.dr. H. Corporaal

Samenstelling promotiecommissie:

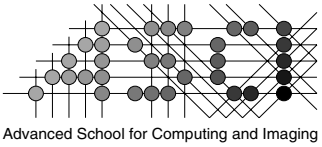
Rector Magnificus,	voorzitter
Prof.dr.ir. A.J. van de Goor,	Technische Universiteit Delft, promotor
Prof.dr. H. Corporaal,	T.U. Eindhoven / IMEC, promotor
Prof.dr.ir. E.F. Deprettere,	Universiteit Leiden
Prof.dr.ir. Th. Krol,	Universiteit Twente
Prof.dr.ir. R.H.J.M. Otten,	Technische Universiteit Eindhoven
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft
Dr. C. Eisenbeis,	INRIA, Rocquencourt

Published and distributed by: DUP Science

DUP Science is an imprint of
Delft University Press
P.O. Box 98
2600 MG Delft
The Netherlands
Telephone: +31 15 27 85 678
Telefax: +31 15 27 85 706
E-mail: DUP@Library.TUDelft.NL

ISBN 90-407-2209-9

Keywords: Compilers, Instruction Scheduling, Register Assignment



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 69.

Printed in The Netherlands

This dissertation is dedicated to the loving memory of my mother

Acknowledgements

This Ph.D. thesis is the result of my research at the Computer Engineering group of the Electrical Engineering department of the Delft University of Technology.

First, I would like to express my gratitude to Henk Corporaal, my supervisor, for his everlasting support, valuable comments and for the numerous discussions we had. In addition, I want to thank prof. Ad van de Goor for being my promotor and for giving me the opportunity to perform my research in his group.

Secondly, I thank the reviewers of this thesis, Andrea Cilio, Henjo Schot and my sister Willemien Korfage for their valuable comments on (parts of) the first drafts of this thesis. I thank Walter Groeneveld for his contribution on the formulation of the "stellingen". Furthermore, I would like to thank my fellow Ph.D. students within the MOVE project: Marnix Arnold, Jeroen Hordijk, Steven Roos and especially Jan Hoogerbrugge for their work on the TTA compiler. I would like to thank the system administrators Jean-Paul van der Jagt, Tobias Nijweide and Bert Meijs for providing an excellent working computer environment. In addition, I would also like to thank all my former colleagues and students of the Computer Engineering group for the enjoyable working environment.

Finally, I would like to thank my family, friends and TNO colleagues for their support and encouragement.

Johan Janssen
Delft, July 2001

Contents

1 Introduction

Today, microprocessors can be found in virtually every electronic device. Not only workstations and PCs contain microprocessors; they can also be found in equipment for daily use such as television sets, mobile phones, PDAs (Personal Digital Assistant), microwaves and cars, or in specialized devices such as the automatic pilot in an aircraft, robots and medical instrumentation. More than 100 million microprocessors for general-purpose computers (PCs, workstations, mainframes, etc.) are sold annually. This is however, only the tip of the iceberg. Over two billion microprocessors are estimated to be sold annually for embedded applications [Leh00]. The embedded microprocessor market is growing, according to Dataquest, from $7.5 billion in 1998 to$ 26 billion by 2002 [FBF+00]. Furthermore, the amount of produced embedded software exceeds the produced general-purpose software by a factor of five [EZ97].

The performance of microprocessors is increasing rapidly. This increase is driven by the demand to execute over and over again, more complex and larger applications. Various architectures are used to deliver the requested performance, like *CISC* (*Complex Instruction Set Computer*) and *RISC* (*Reduced Instruction Set Computer*) processor architectures. At this moment, we are in the middle of the *instruction-level parallelism* (*ILP*) era. The power of ILP processing lies in the ability to execute multiple operations in parallel. It should be obvious that this potentially results in large performance improvements. Various ILP architectures like *superscalar architectures*, *VLIWs* (*Very Long Instruction Word architectures*) and *TTAs* (*Transport Triggered Architectures*) are proposed and implemented. Unfortunately, the ability in hardware to execute multiple operations in parallel, by adding extra resources, does not always lead to a

performance increase. Numerous constraints prevent the efficient usage of the resources of ILP processors. To efficiently utilize the available resources the order of operations in the program code should be rearranged. This process is called *instruction scheduling*. *Register assignment* manages the available high-speed on-chip memory elements called *registers*. These registers are used for holding temporary values produced by the operations. The order, in which register assignment and instruction scheduling are applied, plays an important role in the exploitation of ILP. An efficient register assignment may hinder an efficient reordering of the operations. In addition, efficient instruction scheduling may result in an inefficient use of the registers. In this dissertation, this *phase ordering problem* is discussed, solutions are proposed and results are given.

The research is performed within the MOVE project at the Delft University of Technology. The MOVE project aims at bringing instruction-level parallelism within the reach of *application specific processors* (*ASPs*) in a flexible, scalable and cost effective way. These processors are designed for a specialized task and can be found in all kinds of equipment like TVs, cars, copiers, cameras, etc. To achieve these goals, a new processor architecture is proposed and described. This new architecture is called the Transport Triggered Architecture, or in short TTA [Cor98]. Several processors using this new architecture have been designed and implemented [CvdA93, AHC96, TNO99, VLW00]. The performance of TTA processors highly depends on the quality of the compiler. Therefore, to fully exploit the available ILP provided by TTA processors, research is performed to develop new compiler techniques and strategies to enhance instruction-level parallelism.

In this chapter, the concept of instruction-level parallelism is introduced in Section 1.1. The research goals of this thesis are formulated in Section 1.2. An overview of the remaining chapters of this thesis is given in Section 1.3.

1.1 Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) is the family of processor architectures and compiler techniques that enhances performance by executing multiple operations in parallel. The processors provide the resources to execute operations in parallel. The architecture of an ILP processor allows simultaneously access to the duplicated resources, which improves performance. The question of how much ILP is available in programs is addressed in a number of articles [JW89, Wal91, LW92, TGH92, LW97]. Studies to measure the maximum available ILP have critical shortcomings, however. First, many of these studies assume the presence of infinite processor resources and assume perfect program behavior predictors. In this case, the upper limit is too optimistic. Secondly, these studies do not consider modern or future techniques to enhance ILP. This results in a too pessimistic upper limit. Therefore, these studies are of

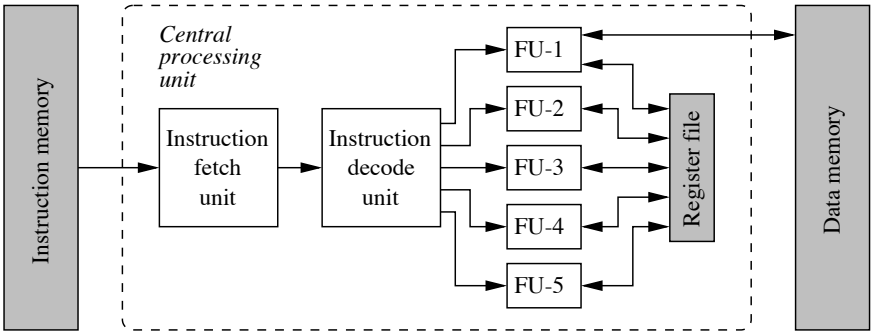


Figure 1.1: General organization of ILP architectures.

limited value. The maximum available ILP estimates range between 2 [JW89] and 1000 [LW97]. One should keep in mind that the exploitable ILP highly depends on the application. Scientific programs have inherently more ILP than control-intensive programs. To discover and to increase the ILP in programs, compiler technology is used. Discovering and exploiting ILP in programs, will be key to future increases in microprocessor performance [SCD+97].

Various architectures used to exploit ILP are described in Section 1.1.1. The architectural trade-off is discussed in Section 1.1.2.

1.1.1 ILP Architecture Arena

The general organization of ILP architectures is shown in Figure 1.1. The *instruction fetch unit* reads the instructions from the *instruction memory*. These instructions are decoded and sent to the *function units* (*FUs*). The *central processing unit* (*CPU*) shown in the figure contains five FUs. The FUs perform the actual computation such as additions, multiplications, etc. One of the FUs in the figure is able to perform load and store operations on the *data memory*. Temporary data is stored in registers. These registers are grouped into the *register file* (*RF*). The FUs exchange data via this *shared* RF.

The main reason for the enormous research interest in ILP architectures nowadays, is the ability to have more silicon space available than a RISC processor requires. This allows the duplication of FUs and data paths. Having duplicated FUs means that multiple operations can be executed simultaneously. The data path transports data between the various resources in a processor. More FUs results in the need for a larger data path. The registers in an RF can be accessed by a limited number of ports. Increasing the number of ports, and thus increasing the data path to and from the RF, enables the exploitation of more ILP.

In [RF93] an excellent overview of the dynamic history of ILP architectures is given. Although the importance of ILP was already recognized in the early

fifties [Wil51], and ILP processors were build in the eighties [BYA93, RYYT89, SS93], it took until the nineties, to become a key technology for microprocessor performance. Rau and Fisher [RF93] classify ILP architectures into three categories: sequential architectures, dependence architectures, and independence architectures.

Sequential architectures

Sequential architectures execute programs that contain no explicit information regarding dependences between operations. The programs for these architectures consist of a sequential operation stream. It is the responsibility of the hardware to detect dependences between operations, and to rearrange the operation order to achieve fast computation; this is called *dynamic scheduling*. An operation starts to execute if it does not depend on an operation currently being executed and if the resources needed for the operation are free.

Implementations of sequential ILP architectures are known as *superscalars* [Joh91]. Superscalars exploit the ILP of a program in hardware; this requires extra logic to detect ILP and to dispatch operations to the FUs. Superscalars differ in their *issue width* (i.e., the maximum number of operations that can be simultaneously executed), and in the complexity of their instruction scheduler. Simple superscalars, like the Alpha 21064, issue operations in the same order as they appear in the program while others, like the PowerPC 601 and the Pentium 4 [Int00], allow instructions to be issued *out-of-order* [SW94]. A disadvantage of superscalars is their limited scalability, for example increasing the issue width often results in a completely new and much more complex design [PJS96].

Superscalars do not require a compiler to exploit ILP, since there is no way to explicitly communicate information regarding ILP from the compiler to the hardware. However, many ILP compiler techniques may be beneficial to enhance superscalar performance [SCD+97, STK98, Wol99].

Dependence architectures

Dependence architectures execute programs consisting of operations *and* information about the data dependences between operations. The programmer or compiler adds this information to the program, which releases the hardware from detecting these dependences. The responsibility of the hardware is to detect operations that are ready for execution and to find free resources.

Data-flow processors [GS95] are representatives of this class. The operations of these processors contain a list of all data dependent successor operations. When an operation finishes execution, a copy of its result is created for each of its successor operations. As soon as all the input operands of an operation and the required resources are available the operation is executed. Since

the operands of an operation are implicitly specified by its predecessors, the operands do not have to be specified.

Independence architectures

Programs for *independence architectures* contain, besides the operations, information about independences between the operations. The compiler is responsible for identifying parallelism in a program. It communicates this information to the hardware by specifying which operations are independent of each other.

An example of an independence architecture is the Horizon architecture [TS88]. The compiler encodes an integer H into each operation. This integer tells the hardware that the next H operations in the operation stream are data independent of the current operation. This releases the hardware from detecting data independence, however, the hardware still is responsible for assigning resources to the operations.

Also the *Explicitly Parallel Instruction Computing* (*EPIC*) architecture [ACM+98] is classified as an independence architecture. The instructions of an EPIC architecture contain multiple operations. Each instruction includes a template, which indicates whether the operations in the instruction are independent. The template also indicates whether the instruction can be executed in parallel with neighbor instructions. An example of the EPIC architecture is the IA-64 instruction set architecture [IA99] as developed by Intel and Hewlett Packard in a joint effort. An actual implementation is Intel's Itanium processor [Abe00].

Another independence architecture is the Very Long Instruction Word (VLIW) architecture [BYA93, DT93, L+93, SS93, GNAB92, HA99, Kla00]. A VLIW compiler not only releases the hardware from detecting independences, but it also assigns the FUs to the operations. A VLIW program specifies on which FU each operation should be executed, and when each operation should be issued. In the context of VLIW architectures, it is important to distinguish between operations and instructions. An operation is a unit of computation, such as an addition, memory load or branch. An instruction consists of multiple operations. The operations in an instruction are issued simultaneously.

The compiler plays an important role to enhance the performance of VLIW processors. In fact, the compiler decides in which order operations are executed, with respect to the data dependence and resource constraints. To accomplish this the compiler uses a detailed description of the processor. It must exactly know how many operations can be executed in parallel. This is in contrast with the previous discussed approaches, where the hardware does the assignment of FUs to operations. Research in the area of exploiting ILP with the use of compilers is still ongoing as a result of the growing interest in VLIW processors. Examples of VLIW implementations are Cydrome's Cydra

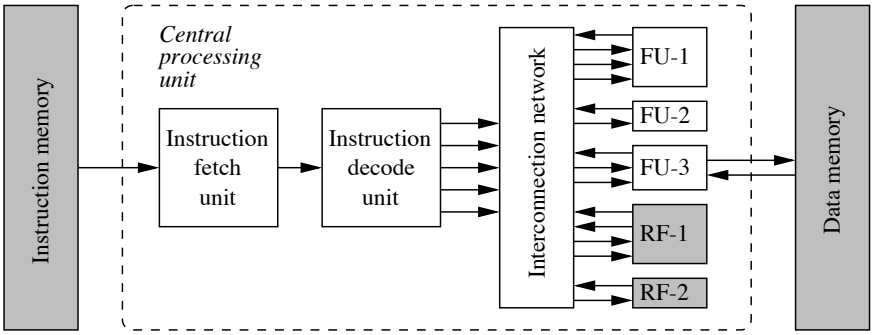


Figure 1.2: General organization of a TTA.

5 [RYYT89], Multiflow's TRACE 14/300 [SS93], Philip's TriMedia [SRD96] processor, the TMS320C6201 DSP processor of Texas Instruments [TI99], BOPS's ManArray [PV00] and Transmeta's Crusoe processor [Kla00].

The Transport Triggered Architecture (TTA) is also classified as an independence architecture. TTAs resemble VLIWs; however, where VLIWs specify operations in an instruction, TTAs also specify the transports between FUs and RFs. This gives the compiler an even larger responsibility, since now not only the FUs and registers need to be assigned but also the transport resources. The general organization of a TTA is given in Figure 1.2. It differs from other ILP architectures (see Figure 1.1) in the sense that not all FUs require a direct connection to the RF. The FUs exchange data via the *interconnection network* instead of using a shared RF. This reduces the complexity of the data path considerably as will be shown in Section 2.1.

The difference, between the three classes of ILP architectures, is the division of the responsibility of the ILP exploitation between the hardware and the compiler. Figure 1.3 summarizes the responsibilities for each type of ILP architecture.

1.1.2 Architectural Trade-off

The most visible ILP processors are general-purpose processors, which can be found in workstations and PCs. For these processors, the superscalar architecture is the current technology of choice. Superscalars have advantages compared to VLIWs and TTAs because they provide binary compatibility, which allows existing applications to run on new machines with a different level of ILP without recompiling. Binary compatibility is an important issue for users who want to upgrade their hardware, without buying new application software. The dynamic scheduling ability of superscalars can adapt the order of execution in situations that were unknown at compile time. It can handle operations with variable latencies, such as load operations with potential cache

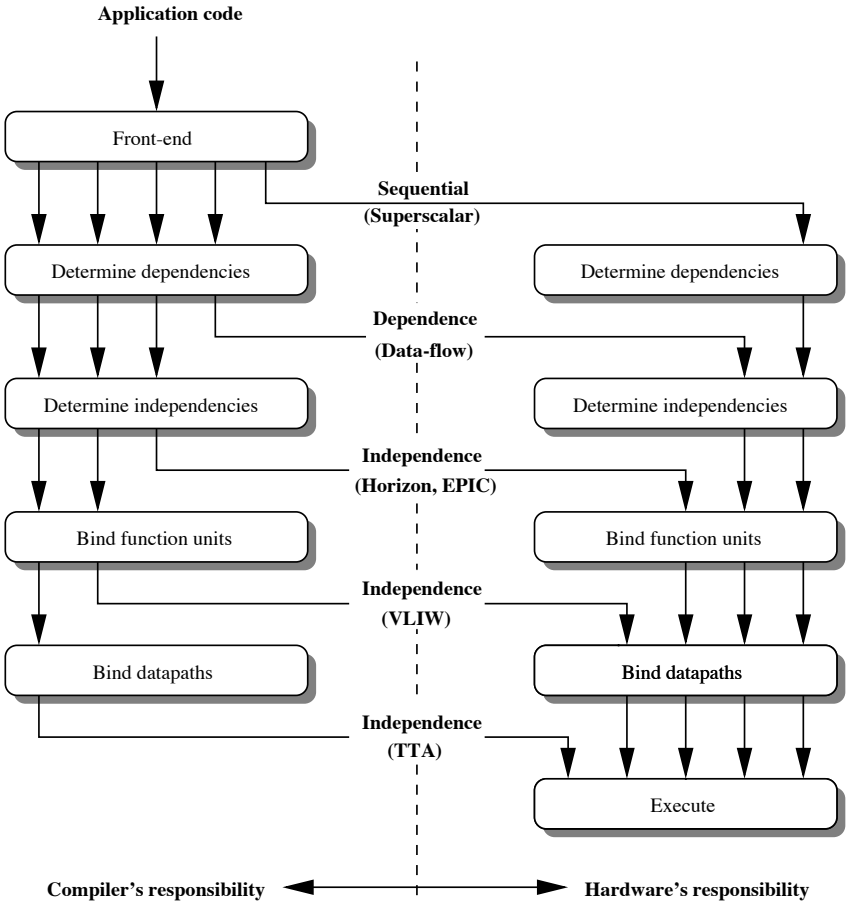


Figure 1.3: ILP architecture classification.

misses. Furthermore, it can reorder memory references whose independence could not be determined at compile time. It does this by comparing their effective addresses. On the other hand, superscalars have a limited view of the operations it can reorder, typically 4 to 64 operations can be considered. The hardware needed to perform the data independence tests and the assignment of the resources largely limits the scalability of superscalars. The number of transistors required to implement this level of intelligence is substantial and the time it takes to execute this work also adds a significant overhead to the pipeline. The enormous design effort required to increase the issue width of a superscalar increases the time-to-market. Furthermore, the increasingly complex design needed to make these processors puts a question mark over how many companies can afford designing them.

Dataflow and Horizon processor architectures are academic research projects and did not found widespread application in the microprocessor market. Processors with an EPIC architecture are just starting to emerge. Initially they will replace superscalars in high-end PCs and workstations. Their hardware complexity is between superscalars and VLIWs. Compiler techniques developed for VLIWs and TTAs are also useful for EPIC based processors.

For Application Specific Processors (ASPs) time-to-market is an important issue. For many companies, it is of vital importance to introduce high quality ASPs at a low cost and within a short time frame. This is the promise of VLIWs and TTAs: by removing complexity from the hardware, simple processors are created that increase performance far more easily than superscalars. Simple hardware increases clock speeds more aggressively than is possible with today's complex superscalars, and more FUs can be easily added to exploit the parallelism existing in applications. TTAs are even more scalable than VLIWs since they do not require that each FU has its own private connection to the RF. VLIWs and TTAs can exploit large amounts of ILP with relatively simple control logic. This not only results in less silicon, but also reduces the power consumption. The compiler for these architectures can reorder the operations within a larger scope, normally tens or hundreds of operations. This gives a major performance benefit compared to superscalars. Unfortunately, VLIWs and TTAs do not provide binary compatibility. Changing the issue width requires recompiling the application. Several methods are proposed to solve this limitation [Rau93, CS95], however no widely accepted solution has been found yet[1]. For companies that develop ASPs, binary compatibility is not an issue, because they usually own the application code and can recompile it for another processor. VLIWs and TTAs are not able to adapt their instruction scheduling strategy to run-time unpredictable situations. When, for instance, a cache miss is encountered and the following instruction needs the data, the processor is locked until the data is available. Furthermore, the instruction scheduler is often hindered by ambiguous memory references. Compile-time analysis can help to alleviate this problem. The instructions of a VLIW or a TTA specify which n operations must be executed in parallel. However, it is very unlikely that n operations can be found. The empty places in the instructions are then filled with no-ops. This lowers the code density compared with superscalars. Efficient encoding and compression techniques [CBLM96, L$^+$93, RYYT89] can be used to solve this problem. Although the market for processors in embedded systems is less visible than the market for superscalars, the embedded market is much larger. Due to these reasons, new compiler techniques for VLIW and TTAs have a lot of interest in academic and industrial research.

[1]The Java [GJS96] platform solves this problem by using a virtual machine. This extra software layer decreases performance and does not yet support the exploitation of ILP. Research has to prove whether ILP can efficiently be exploited by the Java platform [EA97, GV97]. Code morphing [Kla00], as introduced in Transmeta's Crusoe processor translates during execution x86 instruction in VLIW instructions. This software layer implements a virtual machine on a VLIW processor.

1.2 Research Goals

The research described in this dissertation is performed within the MOVE project. This project aims at the automatic generation of application specific processors and their compilers. In this project, a new architecture is developed: the Transport Triggered Architecture (TTA). TTAs are very well suited for ASPs since they provide scalability and flexibility. Since TTAs fall into the category of independence architectures, the compiler is responsible for detecting and exploiting ILP. Two of the most important code generation phases for ILP processors in general, and TTAs in specific, are register assignment and instruction scheduling [HP90]. Applying these two phases separately may have major performance drawbacks. This is especially true for applications for which registers are a critical resource, and for processors with a small register set.

In this dissertation, problems related to the interaction between register assignment and instruction scheduling are analyzed and new methods are researched. This includes the following topics:

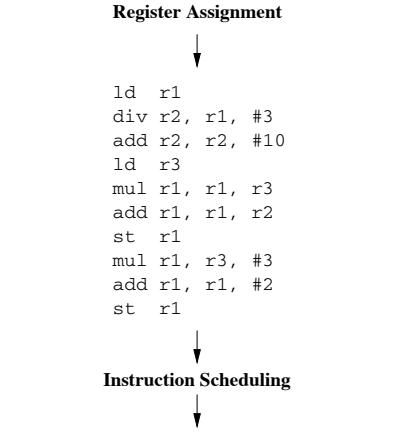
Evaluation of the phase ordering problem
An evaluation of the phase ordering problem of instruction scheduling and register assignment is given. The instruction scheduler is responsible for creating a legal reordering of operations such that the execution time of a program is reduced and the semantics of the program are preserved. The task of the register allocator is to assign the program's variables to the registers of the processor. Instruction scheduling can be done either before or after register assignment. Consider the example in Figure 1.4, which shows three possible scenarios for scheduling the code fragment of Figure 1.4b, assuming a 2-issue processor with three registers. When register assignment is carried out before instruction scheduling (Figure 1.4c), the selection of registers may limit the possibilities to reorder the operations. This has a negative impact on the application's performance. On the other hand, when scheduling precedes register assignment, more variables become live simultaneously. For the scheduled code in Figure 1.4d no legal assignment can be found with three registers. This means that register assignment has to introduce extra operations, so-called spill code, to read and write values from memory. This lengthens the program and increases the execution time. The code example of Figure 1.4e shows the generated code when register constraints and instruction scheduling freedom are considered simultaneously. This approach results in the fastest executing code. In general, the more simultaneously issued operations, the more registers are potentially required. Therefore, it seems advantageous to address register assignment and instruction scheduling simultaneously in order to maximize ILP and to manage the registers efficiently. In this thesis, phase orderings are evaluated and TTA related issues are researched. In addition, solutions proposed in the literature are discussed.

```
for(i = 0; i < 100; i++)
{
  a[i] = a[i]/3 + 10 + a[i]*b[i];
  b[i] = b[i]*3 + 2;
}
```

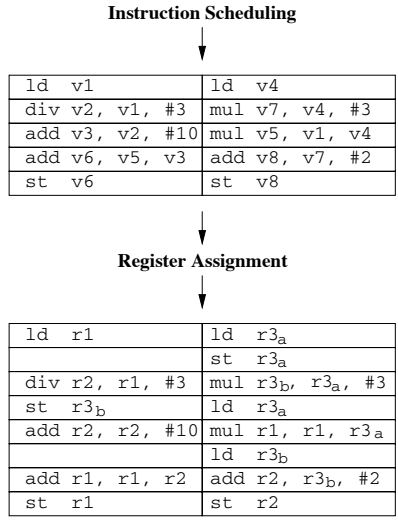
a) Example loop.

```
ld   v1
div  v2, v1, #3
add  v3, v2, #10
ld   v4
mul  v5, v1, v4
add  v6, v5, v3
st   v6
mul  v7, v4, #3
add  v8, v7, #2
st   v8
```

b) Code without loop control.



c) Register assignment before instruction scheduling.



d) Register assignment after instruction scheduling[2]

ld r1	ld r2
div r1, r1, #3	mul r3, r1, r2
add r1, r1, #10	mul r2, r2, #3
add r1, r3, r1	add r2, r2, #2
st r1	st r2

e) Integrated register assignment and instruction scheduling.

Figure 1.4: Motivating example: phase ordering problem.

[2]The live ranges of the variables v4 and v7 are replaced by two short live ranges. Both live ranges with the index *a* originate from the live range of variable v4, while the live ranges with index *b* originate from the live range of v7.

Integrated register assignment and local scheduling
A new algorithm, which integrates register assignment and local scheduling, is developed. Local scheduling is one of the simplest instruction scheduling techniques. It exploits the ILP within basic blocks. A basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and always leaves at the end. It will be shown that the introduced algorithm can gracefully handle situations in which insufficient registers are available. It also inserts store and reload code around procedure calls to preserve the state of the program. The state of the program consists of the contents of the registers. The results of the new introduced method are evaluated and compared with the results of early assignment methods, within the same compiler.

Integrated register assignment and global scheduling
Normally, a basic block consists of a small number of operations. This gives few opportunities to exploit ILP. Therefore, it is beneficial to exploit the ILP across several basic blocks. This increases the register requirements. We research the effectiveness of integrated assignment using a global scheduler, which exploits ILP without being restricted to basic block boundaries.

Integrated register assignment and software pipelining
A powerful and efficient scheduling technique for exploiting ILP in loops is software pipelining. It results in high performance, but increases the register requirements. When this scheduling technique runs out of registers, the compiler is faced with a severe problem. We present a solution based on our developed integrated register assignment and instruction scheduling technique.

Efficient code generation in the context of partitioned RFs
A practical implementation of high performance ILP architectures is constrained by the difficulty to build a large multi-ported RF. A solution is proposed to partition the RF into smaller RFs while keeping the total number of registers and ports equal. The advantages and disadvantages of partitioning RFs are discussed. In this dissertation, compiler techniques are proposed to generate code for TTAs containing partitioned RFs. Solutions for separated register assignment and instruction scheduling, as well as for an integrated approach are presented and the results of experiments are given.

1.3 Thesis Outline

This dissertation is organized as follows. Chapter 2 describes the concept of TTAs. Starting from the VLIW concept this new class of processors is derived. In Chapter 3, the compiler framework is discussed. This includes a description of instruction scheduling and register assignment. A thorough knowledge of compiler techniques is necessary in order to comprehend the new methods. Chapter 4 discusses the environment for the experiments, including the used TTA configurations and benchmarks. The relationship between

instruction scheduling and register assignment is described in Chapter 5. It evaluates techniques from literature that tackle the phase ordering problem. The new developed method, which fully integrates register assignment and instruction scheduling in a single phase, is described in the following three chapters. Chapter 6 describes the simplest case. Register assignment is integrated with a local scheduler. A local scheduler can exploit only a modest amount of ILP; therefore, in Chapter 7, a global scheduling method is used. In Chapter 8, the application of integrated register assignment in combination with software pipelining, an even more aggressive scheduling technique, is discussed. This scheduling technique can only be applied to loops. When the amount of exploitable ILP increases, the register pressure increases also. Consequently, more registers are read and written simultaneously, which requires a large multi-ported RF. However, RFs with a high number of ports are difficult to realize. In Chapter 9, solutions are proposed to solve this problem. Two solutions are implemented: one for a phase ordering in which register assignment precedes instruction scheduling, and one method in which both phases are fully integrated. The last chapter concludes this dissertation, the findings are summarized and suggestions for further research directions are proposed.

2 TTAs: An Overview

Increasing computing power is the subject of many research programs. The need for more powerful processors not only originates from general-purpose computing, but also from dedicated applications. To satisfy the need for more powerful processors, computer researchers develop new computer architectures and architectural features. The most well known ILP (Instruction-Level Parallel) computer architectures are superscalars and VLIW (Very Long Instruction Word) processors. Corporaal [Cor98] developed a new computer architecture, called Transport Triggered Architecture (TTA). The research presented in this dissertation is done in the context of this architecture. Knowledge of the TTA concept is necessary to fully understand all aspects of the research presented in the remainder of this dissertation.

The TTA concept evolved from the VLIW concept. The evolution from VLIW to TTA is described in Section 2.1. The TTA's characteristics are described in Section 2.2.

2.1 From VLIW to TTA

TTAs resemble VLIW architectures; both exploit ILP at compile-time. An example of the data path of a VLIW is given in Figure 2.1. This processor contains five *function units* (*FUs*) connected through a 15-ported register file (RF). An operation, performed by an FU, usually reads two values (operands), manipulates them and produces a single result. Consequently, the RF of a VLIW with K FUs must have $3K$ ports: $2K$ read ports and K write ports.

The data transport bandwidth in a VLIW is proportional with K. As was observed by [Cor98], VLIWs are designed for the worst case; however, it is very

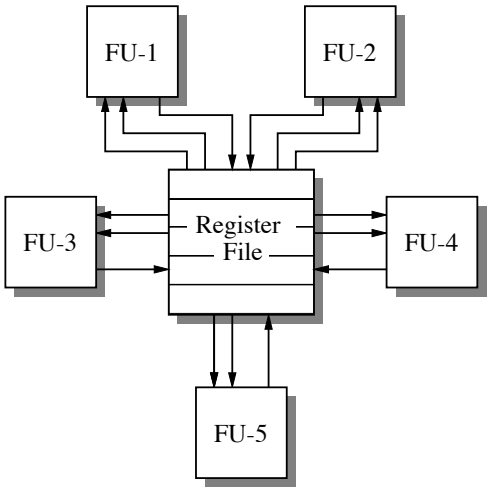


Figure 2.1: Register file connectivity within a VLIW.

unlikely that all FUs are busy simultaneously and that the data bandwidth is utilized for the full 100%. Some VLIWs have more FUs than the instruction size permits [SRD96]. For these VLIWs it is not even possible to keep all FUs busy, assuming single cycle pipelined FUs. Even when all FUs are busy, the data path is not likely to be fully used due to operations that require only one source operand or do not produce a result. The addition of a bypassing network to a VLIW may result in a better performance but decreases the utilization of the data path even further. The complexity of a fully connected bypassing network grows quadratically with the number of FUs [Cor98].

Due to the enormous effort towards the exploitation of more and more ILP, a processor architecture must be designed for scalability. However, the scalability of a VLIW is limited by the rapidly increasing complexity of the required data path; especially as its RF and bypass circuit become complex [Cor98]. This may increase the cycle time of the processor and hence degrades the performance. Furthermore, area and power consumption can become a bottleneck, which can make a processor too expensive and unsuited for specific applications.

Because the data path of a VLIW is rarely used for 100% it seems logical to share the transport capability with other FUs. This not only improves the utilization of the data path, but also decreases the number of ports on the RF. To accomplish this, an extra level of control is needed to ensure that no two transports use the same connection at the same time. This task is the responsibility of the compiler. The generated instructions have to specify transports instead of operations, hence the name of this new architecture: Transport Triggered Architecture.

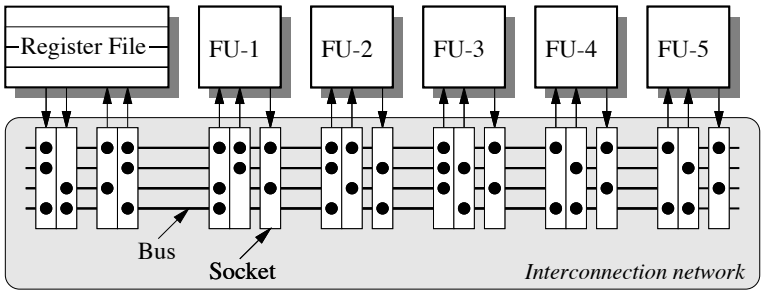


Figure 2.2: Block diagram of a TTA.

2.2 Transport Triggered Architectures

Unlike VLIWs, TTAs do not require that each FU has its own private connection to the RF. An example of the computing core of a TTA is given in Figure 2.2. An FU is connected to the RF by means of an interconnection network. It contains *buses* and *sockets*. A socket can be viewed as a gateway, which is able to pass one data item per cycle. The inputs and outputs of the FUs and RFs are connected to respectively input and outputs sockets. It is not necessary that all buses are connected to a socket. In the figure, the dots indicate to which bus a socket is connected.

FUs can be designed separately, pipelined independently and can have an arbitrary number of inputs and outputs. Examples of standard FUs are:

- *Instruction Fetch Unit:* Reads the instructions from memory and controls the flow of the program (jumps and calls).
- *Integer Unit:* Performs integer operations (add, subtract, etc.).
- *Floating-point Unit:* Performs floating-point operations (add, subtract, etc.).
- *Logic Unit:* Performs logical operations (and, or, xor, etc.).
- *Load/Store Unit:* Reads data from and writes data to external memory.

For some applications it is profitable to have FUs dedicated to a specific task, for instance a Multiply-Add or an RS232 - interface [AHC96]. These *Special Function Units* (*SFUs*) can also easily be integrated within the processor and exploited by the compiler.

The design space of TTAs is enormous. The number and type of FUs and RFs, and the capacity of the interconnection network can be changed easily. Performance gains can be achieved by: adding FUs, pipelining FUs, increasing the number of buses, or changing the RFs. Due to these qualities TTAs are extremely useful for Application Specific Processors (ASPs). For more details

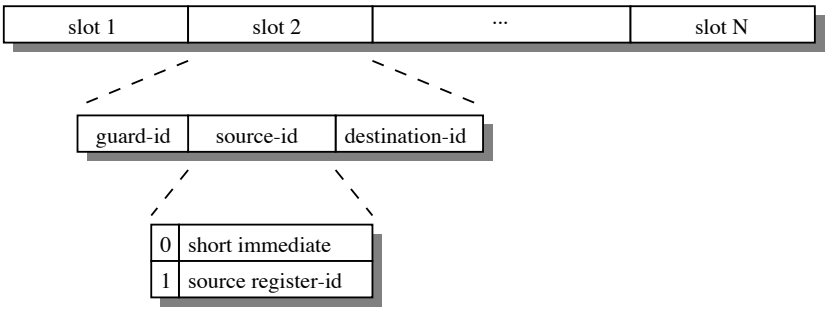


Figure 2.3: General instruction format of a TTA.

on TTAs and prototype realizations, the reader is referred to [CM91, Cor93, CvdA93, AHC96, Cor98, TNO99].

TTAs are composed of various highly regular building blocks. These building blocks can be customized to the needs of an application. In the remainder of this section, a general description of these building blocks is given. In addition, the instruction format and TTA specific properties that allow the exploitation of ILP to a greater extent are discussed.

2.2.1 TTA Instruction Format

TTAs mirror the traditional programming model. Traditional architectures are programmed by specifying operations. The data transports between FUs and RFs are implicitly triggered by executing the operations. Therefore, traditional architectures are called *Operation Triggered Architectures* (*OTAs*). TTAs are programmed by specifying the data transports; as a side effect, the operations are performed. Programming TTAs shows much resemblance with programming VLIWs. Instead of packing the operations in a single instruction, like VLIWs, TTAs pack multiple transports in a single instruction.

The general instruction format of a TTA is shown in Figure 2.3. Each slot of the TTA instruction controls directly a bus. A data transport, also called *move*, describes a data transfer between the source and destination as specified by the source-id and destination-id of a slot. The source- and destination-ids specify registers. Examples of registers are: general-purpose registers, operand registers, trigger registers and result registers. The source-id does not always specify a register; it can also contain a small integer. A 1-bit flag is added to the source-id to specify whether the source-id specifies a register or contains a *short* immediate. The guard-id is used to support conditional or guarded execution. It specifies a Boolean expression. The outcome of this expression determines whether a move is executed or not. In a two-stage instruction pipeline, the FUs do not have a result register. For these TTAs, the destination-id specifies the FU output instead of a register.

2.2.2 Function Units

The function units or FUs are the components of a TTA that perform the computations or communicate with the outside world. Tasks performed by the FUs are for example: additions, multiplications, and load and stores to memory.

An FU contains one or more input and output registers. These registers are subdivided into three types: *operand* registers, *result* registers and a *trigger* register. Executing an operation with n (source) operands consists of moving $n-1$ operands to the operand registers and one operand to the trigger register of the FU. A trigger register is a special kind of operand register because moving an operand to a trigger register starts an FU operation. The results of the operation can be read from its result registers. As an example we show how a RISC type `add` instruction translates into three transports:

```
add r3,r2,r1;  ⇒  r1 → add.o; r2 → add.t;
                  add.r → r3;
```

First, the values of `r1` and `r2` are moved from the RF to the operand and trigger registers `add.o` and `add.t` of the FU. After a delay (depending on the latency of the adder), the result is moved from the result register `add.r` to `r3` in the RF. An FU may support various operations. The type of operation is specified by its trigger register. Writing to a specific trigger register of an FU starts the corresponding operation.

Operations that have a longer latency than a single cycle are subject to pipelining. There are several alternatives for pipelining the FUs. The two alternatives that are currently supported are *hybrid pipelines* and *virtual-time latching* (*VTL*) pipelines.

A hybrid pipeline ensures that data in the pipeline is never overwritten. This implies that when a result is not read from the result register, another operation that is started one cycle later will not overwrite the old result. A pipeline full exception is raised, when an attempt is made to write to a trigger register of a hybrid pipelined FU, while it cannot accept an operation because the pipeline is full. On the other hand, the processor locks when a read attempt is made from a result register, which contains no valid data. The lock is released when the result register receives a valid value from a previous pipeline stage. This pipelining discipline is used in the MOVE32INT [CvdA93] processor and the Phoenix processor [CL95].

Operations executing on a VTL pipelined FU proceed unconditionally from one pipeline stage to the next (except for data and instruction cache misses and exceptions). This means that the value of a result register can be overwritten even when the old value was not even read. The availability of the old value depends on how soon another operation triggers the FU. It is the responsibility of the compiler to ensure that no data is unintentionally overwritten.

The experiments performed in [Hoo96] show that both pipeline alternative have a comparable performance. VTL pipelined FUs are easier to implement in silicon and less likely increase the cycle time due to complex con-

trol logic [Cor98]. In the remainder of this thesis, all FUs have VTL pipelines. For a detailed evaluation of various pipeline disciplines the reader is referred to [Cor98].

2.2.3 Register Files

Registers can be seen as small, first level, fastest accessible components of the memory hierarchy. Registers bridge the gap between main memory speed and the rate at which FUs can process data. The values of variables are stored in registers to speed up the execution time. Registers are grouped into register files (RFs). An RF has a number of ports through which the registers are accessible. The RF in Figure 2.2 has two read ports and two write ports. Two register reads and two register writes can be performed simultaneously. The more ports on an RF the more freedom in access patterns; however, increasing the number of ports increases the chip area [CDN95], the access time [FJC95, Cor98], and the power consumption [ZK97]. The current TTA framework provides three types of RFs: RFs for integer registers (32 bits), RFs for floating-point registers (64 bits) and RFs for Boolean registers (1 bits). However, no fundamental restrictions exist for using an arbitrary number of bits for the registers, or for using an arbitrary number of RFs.

2.2.4 Immediates

Not all values originate from registers. For example, values that are fixed throughout the program can be coded into the instructions. As already discussed in Section 2.2.1, small immediates can be coded in the source-id of a move. The size of a short immediate is usually less or equal to 8 bits.

Immediates that do not fit into the source-id of a move are handled differently. To support these *long* immediates the compiler must encode them in the instructions. This can be done by adding one or more *immediate fields* to the instruction format. When an instruction is fetched from the instruction memory, the immediates stored in the immediate fields are placed in special registers. These registers are called *immediate registers*. The immediates can now be transported by specifying the id of the immediate register in the source-id of the move. Other implementations to handle long immediates are also feasible, see [Jan01].

2.2.5 Move Buses

The communication between FUs and RFs is done over the interconnection network. This network consists of a set of sockets and buses. A bus, also denoted as move bus, is controlled by a slot of the TTA instruction. The implementation of a move bus not only provides the necessary data transport capability,

but it also performs the distribution of the control signals. The control signals include the source and destination register-ids and the signals for locking, guarding and exceptions.

A fully connected interconnection network simplifies the code generation task. From the hardware point of view, a fully connected interconnection network may result in a high bus load, which may affect the cycle time. For ASPs, the interconnection network should represent the communication requirements for the executed application(s) under its performance constraints. Paths in the interconnection network that are heavily used should be provided, while less frequently used paths can be removed[1].

2.2.6 Sockets

The interface between the interconnection network and the FUs and RFs is provided by input and output sockets, see Figure 2.2. Sockets primarily consist of a comparator, input multiplexers (for the input sockets) and output demultiplexers (for the output sockets). The register-id that is supplied on the bus is checked by the socket whether the specified register is accessible through it. When a register is accessible, the data is passed in the wanted direction.

2.2.7 Control Flow and Conditional Execution

To execute high-level language statements, such as while and if-then-else statements, the processor must be able to change the flow of control conditionally. This is usually done by changing the contents of the program counter. In a TTA, the flow of control can be changed by directly writing a value to the program counter. The program counter is accessible for writing through the *jump* register. A jump is simply performed by writing the address of the target of the jump into the jump register. Depending on the instruction pipeline, the jump can have one or more delay slots.

A jump operation usually executes under a certain condition. When this condition is met, the jump is carried out, otherwise normal execution continues. TTAs support conditional execution by means of *guarded* or *predicated execution*. A Boolean expression is associated with each move. Only when this expression is evaluated to be true, the move takes place. The Boolean expression, also called guard expression, is constructed out of Boolean values that are stored in the Boolean RF. These values are defined by compare operations. The following C-code fragment:

```
        if(a > b)
          c = a;
label:
```

[1]The current compiler requires that there should be at least a (single) connection between the RF and the FU registers.

translates with guarded execution into:

```
        r1 → gt_o;  r2 → gt_t;    /* b1 = a > b */
        gt_r → b1;
        !b1:label → jump;         /* if ( b1 == false ) goto label */
        r1 → r3;                  /* c = a*/
label: ...
```

where the variables `a`, `b` and `c` are respectively mapped onto the registers `r1`, `r2` and `r3`. The operation `gt` performs the *greater-than* operation and generates a Boolean value. Boolean register `b1` holds the Boolean value and guards the jump operation. The notation `!b1:` indicates that the operation following this guarded expression is only executed when `b1` evaluates to false.

Conditional execution can also be used to prevent the insertion of jumps into the code. Consider the following example:

```
if(a > b)
  c = a;
else
  c = b;
```

With conditional execution this code fragment translates into:

```
r1 → gt_o;  r2 → gt_t;      /* b1 = a > b */
gt_r → b1;
b1:r1 → r3  !b1:r2 → r3;    /* if ( b1 == true ) c = a else c = b */
```

where the variables `a`, `b` and `c` are respectively mapped onto the registers `r1`, `r2` and `r3`. The notation `b1:` indicates that the operation following the guarded expression is only executed when `b1` evaluates to true.

2.2.8 Software Bypassing

Software bypassing is one of the advantages of TTAs above more traditional architectures. Using software bypassing the compiler can eliminate the need of some RF accesses, see for example the following code fragment:

```
r1 → add.o;    r2 → add.t;    /* r3 = r1 + r2 */
add.r → r3;
r3 → sub.o;    r4 → sub.t;    /* r5 = r3 - r4 */
sub.r → r5;
```

Software bypassing allows the two flow dependent moves (`add.r` → `r3` and `r3` → `sub.o`) to be scheduled in the same instruction, provided that the result of the addition is *directly* written to the operand of the subtraction. This optimization shortens the execution time and saves one RF read. This reduces the RF-port requirements. The scheduled version when applying software bypassing is:

```
r1 → add.o;    r2 → add.t;
add.r → r3;    add.r → sub.o;   r4 → sub.t;
sub.r → r5;
```

When the value of `r3` is not needed anymore, the defining move `add.r → r3` can be removed by *dead-result move elimination*: the result of the addition is directly bypassed to the subtraction. This results in the following code:

```
r1 → add.o;    r2 → add.t;
add.r → sub.o; r4 → sub.t;
sub.r → r5;
```

Dead-result elimination not only reduces the register requirements, but also saves a data transport and an RF write access. The freed resources can be used by other moves, which may lead to higher performance.

2.2.9 Operand Sharing

Since moves are handled individually by the compiler, it is also possible to share an operand move by multiple operations. This is illustrated in the following example, which shows two additions with a common operand (e.g., `r1`).

```
r1 → add1.o;   r2 → add1.t;    /* r3 = r1 + r2 */
add1.r → r3;
r1 → add2.o;   r4 → add2.t;    /* r5 = r1 + r4 */
add2.r → r5;
```

When both additions are executed on the same FU, the second operand move `r1 → add2.o` can be eliminated because the value of `r1` is already present in the operand register of the FU. This results in the following code:

```
r1 → add1.o;   r2 → add1.t;
add1.r → r3;   r4 → add2.t;
add2.r → r5;
```

The optimized version saves a move and an RF access. This optimization can only be applied when the following requirements are met: (1) the value of the common operand is the same, (2) the operations execute on the same FU, (3) the common operand is provided to the FU via the same operand register, and (4) the operand register is not changed by other intervening operations.

3 Compiler Overview

A compiler is used to translate a program into executable machine code. Especially for VLIW and TTA based processors, the compiler plays an important role, because it assigns the resources to the operations and the operations to the instructions. In the remainder of this dissertation, new compiler technologies are described to increase the amount of exploitable instruction-level parallelism (ILP). A thorough knowledge of the internals of the TTA compiler is essential in order to understand the work presented in this dissertation.

This chapter describes the compiler infrastructure to generate parallel TTA code. The developed infrastructure is shown in Figure 3.1. The compiler accepts applications written in a High Level Language (HLL) like C, C++ or Fortran. It translates these applications into an intermediate representation, the sequential TTA code. The sequential TTA code is simulated with as input a data set that is representative for future runs of the application. The simulation is used for: (1) verification of the produced code, (2) obtaining application statistics and (3) obtaining profiling information.

The research presented in this thesis focuses on the TTA compiler back-end. It reads the sequential TTA code, the architectural description and the profiling information. It translates the sequential TTA code to parallel (i.e. scheduled) TTA code for the TTA processor specified in the architectural description. When available, profiling information is used to optimize the scheduling process. The parallel code simulator is used to verify the generated code and to evaluate the results.

In the remainder of this chapter, the front-end and the back-end of the TTA compiler are discussed in more detail. Section 3.1 discusses front-end issues relevant for this thesis. The back-end infrastructure is the focus of Section 3.2.

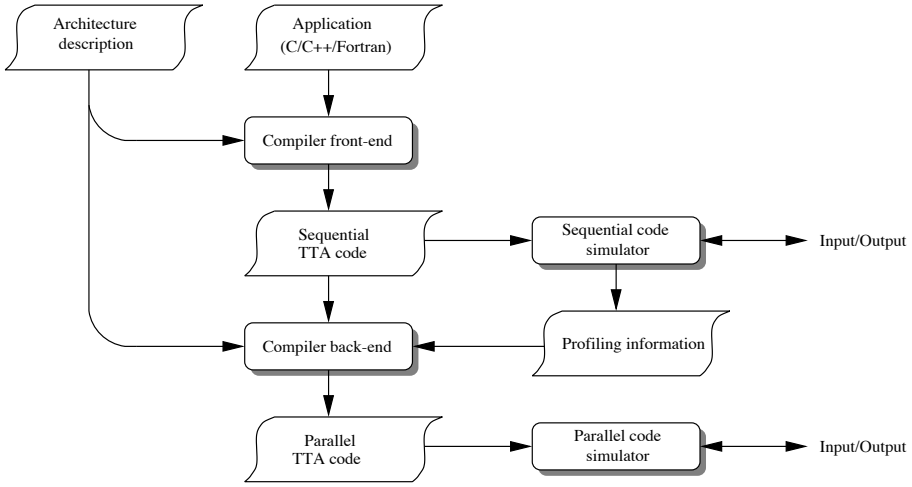


Figure 3.1: Information flow in the TTA compiler.

It describes code restructuring and analysis techniques, necessary to comprehend the methods introduced in this thesis. The two most important phases of the compiler back-end are register assignment and instruction scheduling. In Section 3.3, the basic principles of the popular graph coloring register allocator are given. Finally, Section 3.4 describes instruction scheduling techniques that are used throughout this thesis.

3.1 Front-end

The job of the compiler front-end is to translate the application code written in a HLL into sequential TTA code. In order to be assured of good code quality, good HLL compatibility and a well debugged compiler, the freely available GNU C compiler [Sta94] of the Free Software Foundation is used as the front-end of the TTA compiler. The ported compiler transforms programs coded in the programming languages ANSI C, C++ or Fortran 77 to sequential TTA code. Other compilers can also be used as a front-end. More recently the SUIF C [WFW+95] compiler has been ported to produce sequential TTA code [CC97]. The SUIF C compiler gives more control over optimizations and provides the ability to combine the benefits of exploiting both coarse and fine grained parallelism.

The complete operation repertoire of a generic TTA processor is listed in Table 3.1. Note that no mnemonic is needed for a register copy. A specific TTA processor does not have to support all operations, this is indicated in the table. When operations are not supported by the TTA, the front-end will replace it by other supported operations, or it generates a call to a library function.

Table 3.1: Operation set of the TTA compiler.

Operation type	Mnemonic	Optional
Integer add and subtract	`add`, `sub`	No
Integer multiply and divide	`mul`, `div`, `divu`, `mod`, `modu`	Yes
Word load/store	`ld`, `st`	No
Sub-word load/store	`ldb`, `ldh`, `stb`, `sth`	Yes
Integer compare	`eq`, `gt`, `gtu`	No
Shift	`shl`, `shr`, `shru`	No
Logical	`and`, `ior`, `xor`	No
Sign-extend	`sxbh`, `sxbw`, `sxhw`	Yes
Sub-word insert/extract	`insb`, `insh`, `extb`, `exth`	Yes
Floating-point arithmetic	`addf`, `subf`, `negf`, `mulf`, `divf`	Yes
Floating-point load/store	`ldd`, `lds`, `std`, `sts`	Yes
Floating-point compare	`eqf`, `gtf`	Yes
Type conversions	`f2i`, `f2u`, `i2f`, `u2f`	Yes
Register copy		No

To hold temporary values, variables are used. The front-end uses 128 32-bit integer variables to hold the integer values. When a floating-point RF is available in the target TTA, 128 64-bit floating-point variables are used for storing the floating-point values. When the target TTA does not contain a floating-point RF, the compiler front-end will use integer variables for holding floating-point values. The large number of variable names prevents that the front-end maps variables onto the main memory. The actual register assignment for the target TTA processor is done by the back-end. The front-end uses a single Boolean variable to support conditional execution of jumps. Only jumps are guarded in the sequential TTA code. All moves contain at least one variable or one immediate. The front-end does not apply TTA specific optimizations such as software bypassing.

To support procedure calls, a *call* register is used. Writing to the call register is similar to writing to the jump register (i.e. the program counter), with the difference that the address of the next instruction is placed in the *return address* register. The value of the return address register is used to resume execution when the called procedure finishes execution. The front-end does not insert state preserving code around procedure calls, because register assignment is not performed yet. For interfacing with the operating system a *trap register* is used. The value that is written to it indicates the requested (4.3BSD) system call. Passing information between procedures is accomplished with the use of special variables. These variables are listed below:

1. Variables `v1` and `v2` are used for the stack and frame pointer respectively. These two variables have aliases `sp` and `fp` respectively.

2. Results are returned via `v0` (for integers) and `vf0` (for floating-point numbers). We will use `rv` and `fv` as aliases for `v0` and `vf0` respectively.
3. The integer arguments are passed via variables `v3`..`v6` and the floating-point arguments are passed via `vf1`..`vf4`. The remaining parameters are passed via the stack.

3.2 Back-end Infrastructure

The back-end exploits the ILP of an application and maps it onto the TTA processor that is specified in the architecture description. Figure 3.2 shows the TTA compiler back-end. Transforming the sequential TTA code to correct parallel TTA code requires that the semantics of the application are preserved and the architectural description is respected. In this section, the infrastructure of the back-end is presented. The back-end infrastructure provides the following functionality: (1) I/O components that perform conversions between internal data structures and external files (architecture description reader, sequential TTA code reader, profiling information reader, parallel TTA code writer), (2) analysis functions (control flow analysis, data flow analysis, data dependence analysis) and (3) code transformation functions (function inlining, loop unrolling, grafting and controlled node splitting).

3.2.1 Reading and Writing

The sequential TTA code of the program generated by the front-end is read by the back-end and transformed into an internal representation. The following definitions define the elements of this representation:

Definition 3.1 *A* program $\mathcal{P}$ *describes the behavior of an application. It consists of a set of procedures.*

Definition 3.2 *A* procedure P *is a code abstraction element of a program. Each procedure implements a specific task. Each procedure consists of a set of basic blocks.*

Definition 3.3 *A* basic block b *is a sequence of consecutive instructions in which the flow of control always enters at the beginning and always leaves at the end. A basic block consists of a set of operations.*

Definition 3.4 *An* operation o *describes the computation to be performed on an FU. An operation consists of a set of moves.*

Definition 3.5 *A* move m *describes data transports between hardware components.*

The internal representation of a program $\mathcal{P}$ is annotated with *profiling information*, which consists of the execution counts of the basic blocks and control flow edges. When profiling information has been generated by the sequential

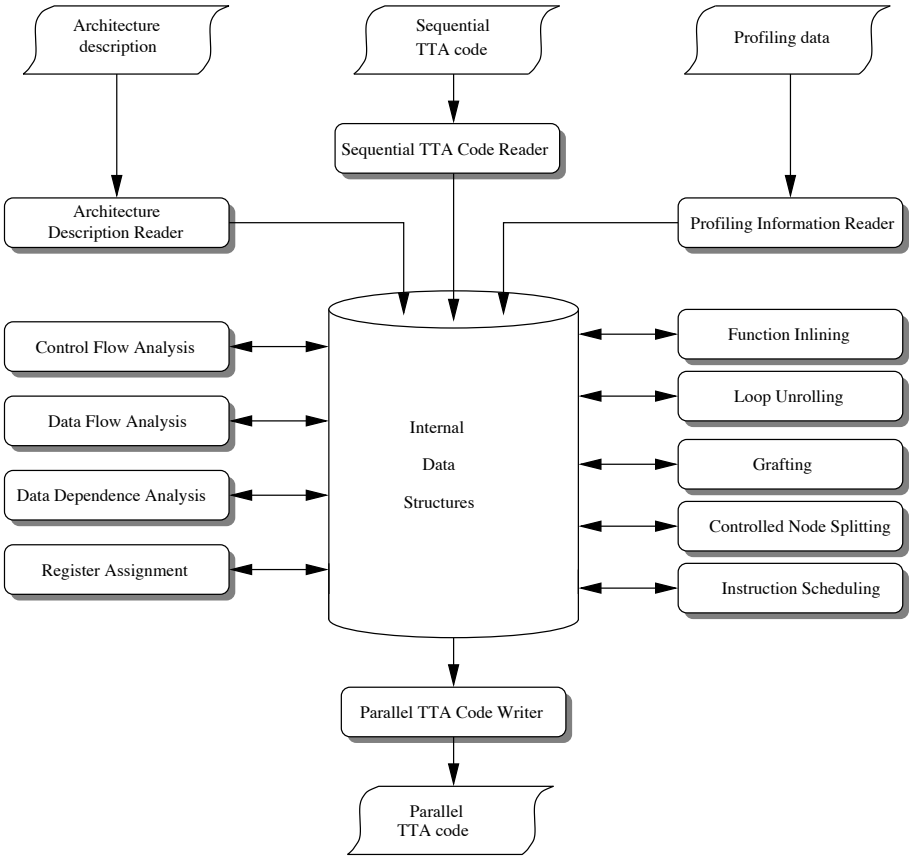


Figure 3.2: Structure of the TTA compiler back-end.

simulator, it is read from a file. Otherwise, it is generated based upon the loop nesting in the procedures.

The machine description file contains the description of the target TTA processor. This information is also stored into an internal representation. The back-end generates parallel code using this internal representation. The generated parallel TTA code is written to a file.

3.2.2 Control Flow Analysis

Conditional branches determine the order in which basic blocks are executed. A *Control Flow Graph* (*CFG*) makes these execution paths visible to the compiler.

Definition 3.6 *The control flow graph CFG of a procedure P is a triple (B, CE, s) where (B, CE) is a finite directed graph, with B the set of basic blocks and CE the set*

of control flow edges. From the initial basic block $s \in B$ there is a path to every basic block of the graph.

Control flow analysis (*CFA*) uses the CFG to compute the relations between basic blocks; successive compiler phases can use this information for optimizing the program. CFA computes the dominator and post-dominator relations between basic blocks, identifies loops, and determines the nesting of the loops [ASU85].

Dominator information is used to identify loops and to determine whether code motion between basic blocks requires code duplication.

Definition 3.7 *A basic block b_i dominates basic block b_j if every path from the initial node of the CFG to b_j goes through b_i. More formally b_i doms b_j.*

Note that a basic block dominates itself. Dominator information is computed by solving control flow equations [ASU85].

Several compiler optimizations require as input a *reducible* CFG. Many definitions for reducible CFGs are proposed. The one adopted here, is given in [ASU85] and is based on the partitioning of the control flow edges into two disjoint sets:

1. The set of *back edges* BE consists of all control flow edges whose heads dominate their tails.
2. The set of *forward edges* FE consists of all control flow edges that are not back edges, thus $FE = CE - BE$.

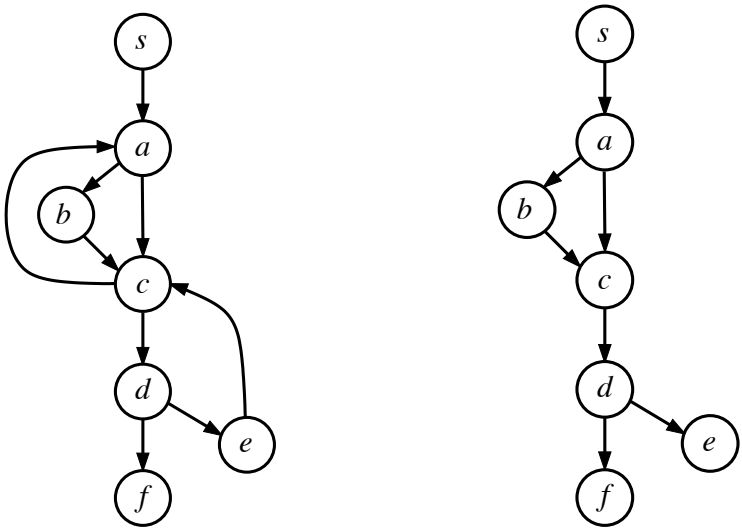
The definition of a reducible flow graph is as follows:

Definition 3.8 *A $CFG = (B, CE, s)$ is reducible if and only if its subgraph $CFG' = (B, FE, s)$ is acyclic and every basic block $b \in B$ can be reached from the initial basic block s.*

The CFG of Figure 3.3a is reducible since $CFG' = (B, FE, s)$ is acyclic, see Figure 3.3b. The CFG of Figure 3.4a is *irreducible*. The set of back edges is empty, because neither basic block a nor basic block b dominates the other. FE is equal to $\{(s,a),(s,b),(a,b),(b,a)\}$, and $CFG' = (B, FE, s)$ is not acyclic.

Many compiler optimizations such as data flow analysis, loop transformations, the exploitation of ILP, and memory disambiguation are simpler, more efficient, or only applicable when the control flow graph of the program is reducible. To overcome this limitation, irreducible CFGs are transformed to reducible CFGs. In the past, some methods were given to solve this problem [CM69, Hec77, ASU85]. Most methods for converting an irreducible CFG are based on a technique called *node splitting*. The principle of node splitting is illustrated in Figure 3.4; basic block a of the CFG is split. The CFG is converted into a reducible CFG at the cost of code duplication.

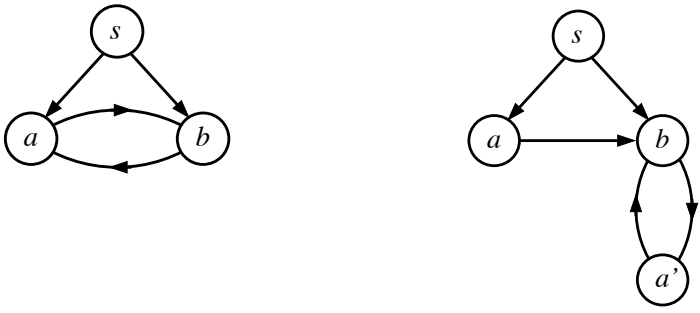
Unfortunately, existing methods have the problem that the resulting code size, after converting an irreducible CFG, can grow uncontrolled. In [JC96] we



a) $CFG = (B, CE, s)$ b) $CFG' = (B, FE, s)$

Figure 3.3: a) Reducible control flow graph; b) the graph $CFG' = (B, FE, s)$.

reported an average increase in code size of 236% for procedures with an irreducible CFG. In the same article, we described a new method for transforming irreducible CFGs to reducible CFGs, called *Controlled Node Splitting* (*CNS*). This method minimizes the number of copies by duplicating only basic blocks with specific properties. This method resulted in an average increase in code size of only 3%. As we have proven in [JC97a] this method is optimal in the sense that it only needs a minimal number of copies to make a CFG reducible. After applying CNS the resulting CFG contains only *natural* loops. Natural loops have one header and may have multiple back edges and exit edges.



a) An irreducible CFG. b) The reducible CFG after applying node splitting to basic block a.

Figure 3.4: An irreducible CFG and its reducible counterpart.

3.2.3 Data Flow Analysis

Data Flow Analysis or *DFA* can be described as the process of ascertaining and collecting information on how a program manipulates data. There are several levels for this analysis. The ones used in the TTA compiler back-end are the analysis at the basic block level and at the procedure level. First, some terms are defined to describe the problem more formally.

Definition 3.9 *A variable v is a place holder for temporary values.*

Definition 3.10 *A variable v is* defined *at a point in a program when a value is assigned to it.*

Definition 3.11 *A variable is v* used *at a point in a program when its value is referenced in an expression.*

Definition 3.12 *A variable v is said to be* live *at a point Q in a program if it has been defined earlier and will be used later. The set $live(Q)$ consists of all variables that are live at point Q.*

Definition 3.13 *The* live range *$lr(v)$ of a variable v is the execution range between its definitions and uses.*

Definition-use chains (*du-chains*) are added between the definitions of a variable and its *reachable* uses. These du-chains are used for memory disambiguation and register assignment.

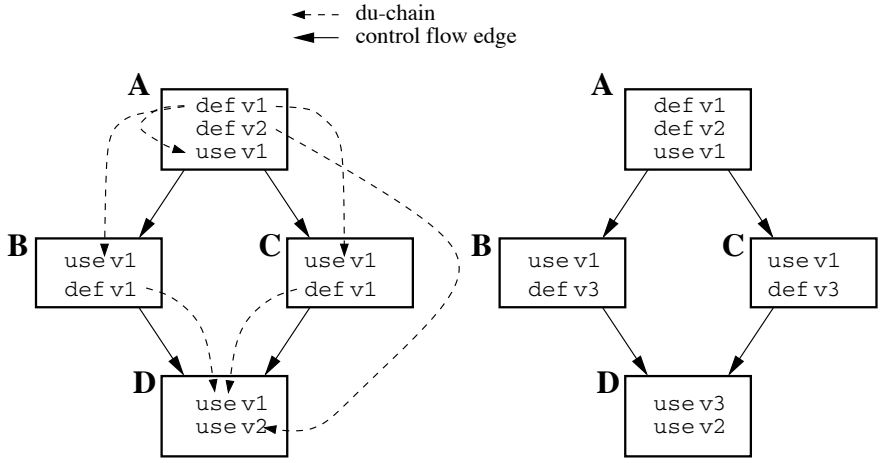
Definition 3.14 *The du-chain(v) of a variable v is a directed graph (N_{du}, E_{du}) that connects the moves that define variable v to the moves that use v. The nodes and edges of a du-chain(v) are defined as:*

$$\begin{aligned} N_{du} &= N_{Def(v)} \cup N_{Use(v)} \\ E_{du} &= \left\{(n_{def}, n_{use}) \mid n_{def} \rightsquigarrow n_{use}, n_{def} \in N_{Def(v)} \wedge n_{use} \in N_{Use(v)}\right\} \end{aligned}$$

where $N_{Def}(v)$ is the set of moves that define variable v and $N_{Use}(v)$ is the set of moves that use variable v. The notation $n_{def} \rightsquigarrow n_{use}$ means that there exists an execution order from n_{def} to n_{use} that does not redefine v.

The du-chains are constructed by applying a standard iterative data flow algorithm [ASU85]. Figure 3.5a shows the du-chains of the variables `v1` and `v2`. In this figure `def vi` denotes a definition of a variable `vi` and `use vi` denotes a use of a variable `vi`.

Renaming is a transformation that may increase the scheduling freedom. The naming of the variables as shown in Figure 3.5a prevents, for example, that the instruction scheduler can reorder the definition and the use of variable `v1` in basic block **B**. Reordering these operations would result in incorrect program execution. This ordering constraint, or dependence, is caused by the re-use of



a) Du-chains of v1 and v2. b) Renaming.

Basic block	$live_{Def}$	$live_{Use}$	$live_{In}$	$live_{Out}$
A	{v1,v2}	$\emptyset$	$\emptyset$	{v1,v2}
B	{v3}	{v1}	{v1,v2}	{v2,v3}
C	{v3}	{v1}	{v1,v2}	{v2,v3}
D	$\emptyset$	{v2,v3}	{v2,v3}	$\emptyset$

c) The live sets of figure b.

Figure 3.5: Data flow analysis.

variable v1. Renaming [CF87, Muc97] removes this type of dependence by splitting up the du-chain. The newly created du-chains are assigned to new variables. In Figure 3.5b, the second live-range of variable v1 is renamed to variable v3. This removes the false dependences in the basic blocks **B** and **C**.

Live-Variable Analysis computes whether a variable is *live* in a particular basic block. For each basic block the sets $live_{Use}(b)$ and $live_{Def}(b)$ are computed. The set $live_{Use}(b)$ is defined as the set of variables that are used in basic block b before they are defined in basic block b. The set $live_{Def}(b)$ is defined as the set of variables that are defined in basic block b before they are used in this basic block. The set of variables that are live on entry and exit of a basic block can now be computed with:

$$live_{In}(b) = (live_{Out}(b) - live_{Def}(b)) \cup live_{Use}(b) \tag{3.1}$$

and

$$live_{Out}(b) = \bigcup_{b' \in Succ(b)} live_{In}(b') \tag{3.2}$$

where $Succ(b)$ is the set of all successor basic blocks of basic block b in the CFG. More formally:

$$Succ(b) = \{b' \in B | \, (b, b') \in CE\} \tag{3.3}$$

Algorithms to solve these equations can be found in [ASU85, Muc97]. In Figure 3.5c the result of applying the live-variable analyses to the program of Figure 3.5b is given. Live information is used by the scheduler to test for off-liveness during speculative execution, and by the register allocator to determine the live ranges of the variables.

3.2.4 Data Dependence Analysis

Data Dependence Analysis (*DDA*) is a vital tool in instruction scheduling, it determines the ordering (also known as data dependence) relations between operations that must be satisfied for the code to execute correctly. The set of relations is represented by a directed graph, called the *data dependence graph* (*DDG*). For each procedure a DDG is build. The scheduler uses the DDG to exploit the available ILP without violating the data dependences.

Definition 3.15 *A* $DDG = (N_{DDG}, E_{DDG})$ *is a finite directed graph, with* N_{DDG} *the collection of operations and* E_{DDG} *the collection of data dependence edges. An edge leading from node* n_i *to node* n_j *indicates that* n_i *must be executed before* n_j*.*

Data dependences between operations indicate accesses to the same location. The set of data dependences E_{DDG} can be divided into three subsets:

- *Memory data dependences* are caused by accesses to the same memory location. The set of memory edges, E_{Mem}, is the set of edges that represents memory data dependences.
- *Register data dependences* are caused by accesses to the same register. The set of register edges, E_{Reg}, is the set of edges that represent register data dependences. These dependences arise when the variables are mapped onto the registers of the target machine.
- *Variable data dependences* are caused by accesses to the same variable. The set of variable edges, E_{Var}, is the set of edges that represent variable data dependences. These dependences arise when the variables are not yet mapped onto the registers.

A data dependence between two operations o_1 and o_2 arises from the flow of data between both operations. Assuming that o_1 occurs before o_2 then the types of data dependences that constrain the execution order are:

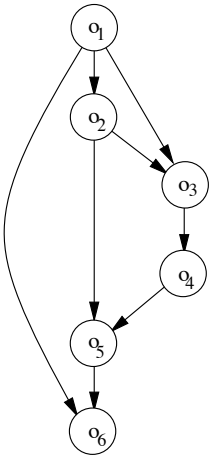
- *Flow dependence*: the value of a memory location, register or variable defined by o_1 may be used by o_2. This dependence is denoted as $o_1\ \delta^f\ o_2$.
- *Anti dependence*: the value of a memory location, register or variable used by o_1 may be redefined by o_2. This is denoted as $o_1\ \delta^a\ o_2$.
- *Output dependence*: the value of a memory location, register or variable defined by o_1 may be redefined by o_2. This is denoted as $o_1\ \delta^o\ o_2$.

```
for(i = 0; i < 100; i++, j++)
     x[i] = a * x[i] + b / y[j];
```

a) Source code.

```
o1  ld   v3, 0(v1)
o2  mul  v4, v7, v3
o3  ld   v3, 0(v2)
o4  div  v5, v8, v3
o5  add  v6, v4, v5
o6  st   v6, 0(v1)
```

b) Code of loop body.



c) The DDG of the loop body.

Flow dependences	$o_1\ \delta^f\ o_2$
	$o_3\ \delta^f\ o_4$
	$o_2\ \delta^f\ o_5$
	$o_4\ \delta^f\ o_5$
	$o_5\ \delta^f\ o_6$
Anti dependences	$o_2\ \delta^a\ o_3$
	$o_1\ \delta^a\ o_6$
Output dependences	$o_1\ \delta^o\ o_3$

d) The dependence relations.

Figure 3.6: Example of dependence relations.

For a correct execution of the application, all three data dependence types must be respected during instruction scheduling. Unfortunately, data dependences may hinder the exploitation of ILP. Avoiding or eliminating data dependences may increase the exploitable ILP. Flow dependences cannot be eliminated and are also known as *true dependences*. Anti and output dependences are known as *false dependences*. They are caused by name conflicts and can be eliminated by renaming [HP90].

Figure 3.6b shows the RISC style code of the loop of Figure 3.6a. The corresponding DDG is given in Figure 3.6c and the data dependence relations with their types are listed in Figure 3.6d. The data dependences can be categorized in the following sets:

$$
\begin{aligned}
E_{Var} &= \{o_1\ \delta^f\ o_2,\ o_3\ \delta^f\ o_4,\ o_2\ \delta^f\ o_5,\ o_4\ \delta^f\ o_5,\ o_5\ \delta^f\ o_6,\ o_2\ \delta^a\ o_3,\ o_1\ \delta^o\ o_3\} \\
E_{Mem} &= \{o_1\ \delta^a\ o_6\} \\
E_{Reg} &= \emptyset
\end{aligned}
$$

Note that the set E_{Reg} is empty because the registers are not yet assigned to the variables.

In the TTA compiler back-end, the nodes of the DDG represent moves instead of operations; edges represent data dependences between moves. The moves that make up an operation must be executed in a specific order. The operand move must be executed before or at the same time as the trigger move. The result move must be executed after the trigger move. To guarantee that the execution order of the moves is not violated, extra TTA specific dependence edges, so-called *intra operation edges*, are added to the DDG.

- *Trigger-result dependence*: guarantees that the trigger move of an operation is always scheduled in an earlier instruction than the result move of the same operation. This relation is denoted with δ^{tr}.
- *Operand-trigger dependence*: guarantees that the operand move of an operation is never scheduled in a later instruction than the trigger move. This relation is denoted with δ^{ot}.

A *delay* is associated with each dependence edge. This delay is added to the data dependence relation in the form of $o_1\ \delta^{type}_{delay}\ o_2$, where $type$ represents the type of the data dependence. The delay indicates that operation o_2 should be scheduled at least $delay$ instructions after the instruction of o_1. For TTAs the delay of a flow dependence caused by a register or variable is always zero (δ^f_0), because values can be defined and used in the same instruction by using software bypassing. This does not apply to flow dependences caused by accesses to the same memory location. Their delay is set to one cycle (δ^f_1) in order to ensure that the correct value is read from memory. In the TTA processor model, it is assumed that when a read and a write to the same location occur in the same instruction, the read will get the previous value. Consequently, anti dependences have a delay of zero (δ^a_0). The delay of an output dependence is one instruction (δ^o_1), because multiple writes to the same register or memory location in the same instruction are undefined. The delay associated with the operand-trigger dependence is at least zero. In the remainder of this thesis, it is assumed that this delay is always equal to zero (δ^{ot}_0). The delay between the trigger and the result is equal to the latency of the FU that will execute the operation ($\delta^{tr}_{delay\ FU}$).

The dependence edges of the DDG impose a partial order in which the moves must be executed. A path in the DDG is called a *dependence path*. The longest dependence path is called the *critical path*. The dependences restrict the reordering of the moves. As already mentioned, reducing the number of dependence edges results in a larger scheduling freedom and may result in faster executing code. Methods for reducing the number of data dependences are register renaming as discussed in Section 3.2.3 and memory reference disambiguation.

Memory reference disambiguation is used for proving the independence between two memory references. It attempts to answer the question: given two memory references m_1 and m_2, could they possibly refer to the same memory location? The most simple memory reference disambiguator answers this

question simply with yes. This preserves program semantics, but limits the exploitation of ILP. A more accurate memory reference disambiguation improves the ability for ILP exploitation by removing spurious data dependences. When the memory disambiguator cannot guarantee whether two memory references never refer to the same location, dependence is assumed.

3.2.5 Loop Unrolling, Function Inlining and Grafting

A method to reduce the overhead of executing loops and to improve the effectiveness of other optimizations, such as the exploitation of ILP is *loop unrolling*. Loop unrolling replaces the body of a loop by several copies of the loop and adjusts the loop-control code accordingly. This enlarges the loop body and decreases the number of iterations of the loop. A larger loop body allows a more aggressive exploitation of ILP, which benefits performance. To take full advantage of loop unrolling in the context of exploiting ILP, variable renaming is applied to the replicated loop bodies. This increases the register pressure.

Function inlining replaces calls to procedures with copies of their bodies. This transformation removes barriers between procedures and increases the scope of optimizations, such as common subexpression elimination and constant propagation. Because the optimization scope is enlarged, more ILP can be exploitable. This has also its consequences for register assignment. More variables are live simultaneously and thus the register pressure increases. Another advantage is the reduction of overhead around procedure calls. A detailed analysis concerning procedure inlining can be found in [HC89]. They claim that 59% of the procedure calls can be eliminated with a modest code expansion (17%).

Basic blocks are often too small too contain sufficient ILP. *Grafting* is a technique that increases the size of basic blocks by duplicating *join-point* basic blocks[1]. In Figure 3.7 the join-point basic block **D** is duplicated. After duplication, the basic blocks **B** and **D**, and **C** and **D'** can be merged into larger basic blocks.

3.3 Register Assignment

Register assignment is the problem of finding a mapping of program variables to the registers of the target machine, while preserving the semantics of the program. A valid solution is to place each variable in a different register. However, usually the number of variables exceeds the number of registers. Fortunately, not all variables are live simultaneously. This means that we can map multiple variables to a single register, as long as the corresponding live ranges do not overlap. A proper register assignment is a mapping such that no register is

[1] Grafting is closely related to tail duplication. Grafting duplicates only join-point basic blocks, while tail duplication copies the complete tree below the join-point basic block.

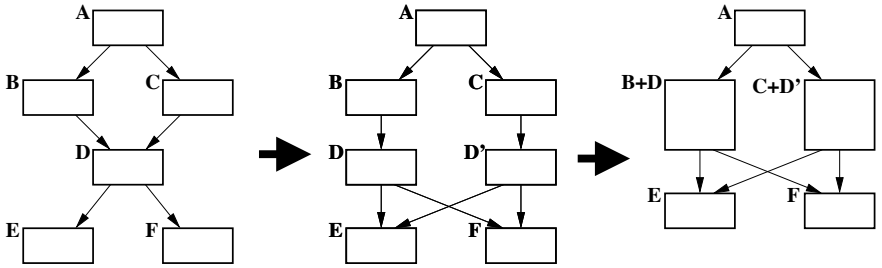


Figure 3.7: Grafting and merging of basic blocks.

assigned to any two variables that are live simultaneously. When no proper register assignment can be found, some variables must be mapped onto memory locations.

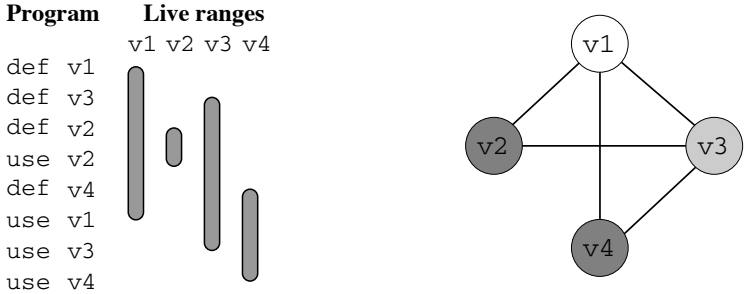
In literature, register allocation is often used when register assignment is meant. Register allocation deals with the determination of type and number of resources, while register assignment is the assignment to register instances [MLD92]. Register assignment can be applied to expressions, basic blocks (local register assignment), procedures (global register assignment), or collections of procedures (inter-procedural register assignment). Aggressive techniques to exploit ILP operate on whole (or even beyond) procedures. Therefore, we believe that at least global register assignment is required to support the exploitation of ILP. In the remainder of this thesis, we mean by register assignment, global register assignment, unless stated otherwise.

3.3.1 Graph Coloring

Graph coloring [BCKT89, Bri92, BCT94, CK91, CAC+81, Cha82, CH90, GL95, GSS89, Mue92] is the most popular method to assign registers. Graph coloring as originally proposed by Chaitin [CAC+81, Cha82] performs register allocation and assignment at the same time. An *interference graph* is constructed to find an efficient mapping. The interference graph consists of a node for each variable and edges between any two nodes, if and only if, the two variables associated with the nodes are live simultaneously for the given operation order. When variables are live simultaneously, we say that these variables *interfere*. Two variables interfere if one of them is live at a definition point of the other [CAC+81].

Definition 3.16 *An interference graph* $IG = (N_{var}, E_{interf})$ *is a finite undirected graph, with* N_{var} *the set of variables and* E_{interf} *the set of interference edges.* E_{interf} *is defined as* $E_{interf} = \{(v_i, v_j) \mid v_i, v_j \in N_{var} \wedge v_i \neq v_j \wedge v_j \in interf(v_i)\}$ *where* $interf(v_i) = \{v \in live(Q) \mid Q \in N_{Def}(v_i)\}$.

The problem of register assignment can be described as: the problem of finding a proper node coloring for the interference graph with k colors, where



a) Program and live ranges. b) Interference graph.

Figure 3.8: Graph coloring example.

k is the number of available machine registers. In a proper node coloring, no adjacent nodes have the same color (register). A graph is said to be k-colorable when a proper coloring can be found. In Figure 3.8a a program and its corresponding live ranges, are given. Figure 3.8b shows the interference graph. Variables `v2` and `v4` can be mapped onto the same register, but `v2` and `v1` cannot. A proper register assignment for the program-segment of Figure 3.8a requires at least three registers.

The first global register allocator, based on graph coloring, was implemented at IBM Yorktown [CAC+81] by Chaitin *et al*. The problem of finding a proper node coloring is known to be NP-complete [Kar72]. Therefore, this register allocator, also denoted as the Yorktown Allocator, uses a fast and simple heuristic. It relies on the following graph theoretic property:

Theorem 3.17 *Given a graph $G = (N, E)$ and a node $n \in N$ such that $degree(n) < k$, then G is k-colorable if and only if $G - n$ is k-colorable. The degree of a node is defined as the number of its neighbors.*

This theorem states that there will always be at least one color left for n, no matter how the reduced graph $G - n$ is colored. When a graph is not k-colorable, some program variables must be placed in a non-register resource. This resource is usually an off-chip read/write memory. This creates a definite speed penalty, so variables must be chosen carefully for these locations.

The Yorktown Allocator uses Theorem 3.17 to simplify the interference graph by repeatedly removing nodes with a degree less than k until the graph is empty, or until only nodes with a degree greater or equal to k are left. If the reduced graph is empty, then finding a k-coloring of the interference graph is reduced to finding a k-coloring of the empty graph. The nodes are then reinserted into the graph in the reverse order in which they were removed. Each node is given a color distinct from its neighbors. Since each node had a degree less than k when removed, each node is guaranteed to be colorable when it is re-inserted. If the reduced graph was not empty, i.e., all nodes of the reduced graph have a degree larger or equal to k, then the graph cannot be colored with

Algorithm 3.1 OPTIMISTICREGISTERALLOCATOR(P)

```
spilling = TRUE
WHILE spilling DO
  IG = BUILD(P)
  SPILLCOSTS(P)
  SIMPLIFY(IG)
  SELECT(IG)
  spilling = INSERTSPILLCODE(P, IG)
ENDWHILE
ASSIGNREGISTERS(P, IG)
GENERATESTATEPRESERVINGCODE(P)
```

k colors and hence the number of registers is insufficient to hold all variables. In the Yorktown Allocator, this situation is solved by marking an uncolorable node for *spilling* and removing this node. The process of simplifying and marking nodes for spilling continues until the graph is empty. Afterwards the variables of the marked uncolorable nodes are spilled; a store to main memory is inserted after each definition of the variable and a load from main memory is inserted before each use. Because the inserted spill code changes the register requirements, the entire process of building, simplifying and spilling is repeated until no further spilling is needed. The resulting interference graph is guaranteed to be k colorable. At this time registers are assigned to variables.

Briggs *et al.* [BCKT89] developed an improvement to the Yorktown Allocator; this allocator is called the Optimistic Allocator. It removes also uncolorable nodes, without marking them for spilling, from the graph as if their degree were less than k. It optimistically hopes a color will be available during the coloring phase. For nodes with degree greater or equal to k it is possible to find a color, when two or more non-interfering neighbors have received the same color. Since more nodes can be colored, less spill code is required. When all neighbor nodes have been assigned all k colors, the node is marked for spilling. Then, just as in the Yorktown Allocator, the variables associated with the marked nodes are placed in memory. In [BCT94], it was reported that the Optimistic Allocator inserts 32% less spill code than the Yorktown Allocator does. Spill code may increase the execution time of a program, therefore it should be avoided when possible.

The register allocator used in the TTA compiler back-end is based on the Optimistic Allocator, see Algorithm 3.1. The phases of this allocator are:

BUILD This phase constructs the interference graph of a procedure P.

SPILLCOSTS In preparation for coloring, a spill cost estimate is computed for every live range. The spill cost per live range reflects the expected increase in execution time when spilling the variable associated with the live range.

Table 3.2: Mapping special variables to registers.

integer

variable	rv	sp	fp	v3	v4	v5	v6
register	r0	r1	r2	r3	r4	r5	r6

floating-point

variable	fv	fv1	fv2	fv3	fv4
register	f0	f1	f2	f3	f4

SIMPLIFY This phase removes nodes with degree $< k$ from the interference graph. Whenever it discovers that all remaining nodes have degree $\geq k$, it chooses a spill candidate. This node is also removed from the graph, hoping a color will be available in spite of its high degree.

SELECT Colors are selected for nodes. The nodes are reinserted in the interference graph in the reverse order in which they were removed, and given a color distinct from their neighbors. Whenever it discovers that it has no color available for some node, it leaves the node uncolored and continues with the next node.

INSERTSPILLCODE Spill code is inserted for the live ranges of all uncolored nodes. If no spill code is required, this phase sets the variable $spilling$ to false otherwise to true. When spilling is required, the register allocator starts again with the first phase in an attempt to color the modified code with k colors.

ASSIGNREGISTERS The registers (colors) are assigned to the variables.

GENERATESTATEPRESERVINGCODE Generates the code required to ensure correct execution around procedure calls.

The preceding discussion assumes that each variable can be mapped on an arbitrary register. However, some variables are mapped on specific registers; an example is the stack pointer, see Section 3.1. These variables and the register they are mapped on, are listed in Table 3.2. The variables that must be mapped on a special register are assigned first. The compiler front-end guarantees that these registers can be assigned correctly and thus do not overlap.

An efficient register assignment is a crucial problem in modern microprocessors. The increasing gap between the internal clock and memory latency requires that variables are kept in registers and to avoid spilling. Furthermore, reads and writes to memory consume more power than a register access. Unfortunately, an optimal coloring of the interference graph does not necessarily correlate with good machine utilization. This will be demonstrated in Chapter 5.

```
v3 → sub.o; #10 → sub.t;
sub.r → v1;
```

a) Original operation sequence.

```
v3 → sub.o;  #10 → sub.t;
sub.r → v1;
fp → add.o;  #offset → add.t;
add.r → v2;
v2 → st.o;   v1 → st.t;
```

b) Spill code sequence.

Figure 3.9: Spilling.

3.3.2 Spilling

A processor has a limited number of registers. When not enough registers are available to hold all variables, some of them are spilled to memory. Spilling is required when the register pressure, at some point in the program, is higher than the number of available registers. The *register pressure* at a point in a program is the number of variables that could reside in registers.

A spilled variable is written to a memory location. Each procedure claims memory locations where it can store the spilled values [ASU85]. This is accomplished by using the frame pointer `fp`. The address of the frame pointer is supplied by the calling procedure. The memory address of a spilled variable is an offset from the *frame pointer*. The basic TTA template does not support loads and stores with offsets, therefore additions are inserted to compute the memory address[2]. The operation sequences of spilling and reloading of a variable `v1` are given in Figure 3.9 and Figure 3.10.

Figure 3.9a shows the TTA code for a subtraction. The result of the subtraction is stored in variable `v1`. When insufficient registers are available, the value of variable `v1` must be spilled to memory. To accomplish this, two operations (an addition and a store operation) are inserted in the code. This is shown in Figure 3.9b. The addition computes the memory address of the memory location, in which the value of `v1` will be stored. It adds an offset to the frame pointer `fp`. The store operation stores the value of `v1` in memory.

A similar situation arises when a value must be reloaded from memory. Assume that the variable `v1` in Figure 3.10a is spilled to memory. In order to retrieve its value, reload code (an addition and a load operation) is inserted in the code. This is shown in Figure 3.10b. The addition computes the memory address in the same way as for spilling. The load operation reads the value from memory and writes it to variable `v1`.

Spilling splits a long live range in multiple shorter live ranges. It is not permitted to spill these newly created live ranges. Spilling these variables will result in an infinite loop in the register allocator. Short live ranges in the original code may also lead to superfluous spill code insertion. Spilling these short live ranges does not reduce the register pressure, because the spill and reload code itself also requires registers. Precautions are taken too avoid the spilling of these short live ranges as much as possible.

[2]When the offset is zero no additions are needed.

```
v1 → mul.o; #5 → mul.t;
mul.r → v2;
```

a) Original operation sequence.

```
fp → add.o; #offset → add.t;
add.r → v3;
v3 → ld.t;
ld.r → v1;
v1 → mul.o; #5 → mul.t;
mul.r → v2;
```

b) Reload code sequence.

Figure 3.10: Reloading.

A number of heuristics that attempt to minimize the impact of spill code insertion when using graph coloring [Cha82, BGM+89] have been published. The spill heuristic used in the register allocator of the TTA compiler back-end is based on the work of Chaitin. However, two changes were made. First of all, we do not use the nesting of loops as an estimate for the execution frequency, but instead profiling information is used. This results in more accurate estimates. Secondly, Chaitin's heuristic assumes that an operation executes in a single machine cycle. However, loading data from memory may take considerable more time than storing data to memory. Our cost function considers this difference:

$$C_{spill}(v) = \frac{\sum_{n_{use} \in N_{Use(v)}} (L(\text{add}) + L(\text{ld})) \cdot f(n_{use})}{degree(v)} + \frac{\sum_{n_{def} \in N_{Def}(v)} (L(\text{add}) + L(\text{st})) \cdot f(n_{def})}{degree(v)} \quad (3.4)$$

where v is a variable that is candidate for spilling, $L(\text{add})$, $L(\text{ld})$, $L(\text{st})$ are respectively the latencies for an addition, a load and a store operation. The expressions $f(n_{use})$ and $f(n_{def})$ are the frequencies of respectively the uses and definitions of v, and $degree(v)$ is the degree of the node v. The heuristic selects the variable v with the lowest cost $C_{spill}(v)$ for spilling.

3.3.3 State Preserving Code

When a program executes a procedure call statement, the state of the calling procedure (*caller*) may be destroyed by the called procedure (*callee*). The state consists of the values residing in registers and the value of the program counter. A convention can be adopted that specifies which registers may be overwritten by a called procedure; however, this practice removes resources that otherwise could be well used. An alternative convention saves the values needed in memory before they are overwritten. When control is returned to the calling procedure, the state is restored from memory and execution can resume. Using

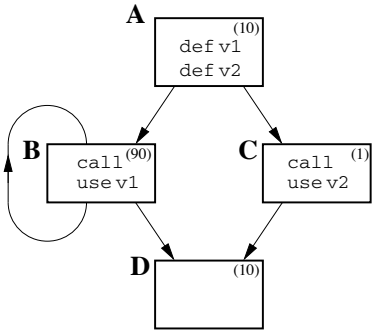


Figure 3.11: Heuristic for caller-saved and callee-saved registers.

this alternative, two choices exist: (1) the caller saves the values before transfer of execution and restores them upon return or, (2) the callee saves them immediately upon entrance and restores them before exit.

To generate high quality code, compilers divide the set of machine registers in caller-saved ($R_{\textit{caller-saved}}$) and callee-saved registers ($R_{\textit{callee-saved}}$). Caller-saved registers are stored and reloaded by the caller and callee-saved registers are stored and reloaded by the callee. Note that only the registers referenced in the procedure need to be saved.

The task of the register allocator is to decide to which set, the caller- or callee-saved set, the variable must be assigned in order to obtain fast code. Consider the CFG in Figure 3.11. The number between parentheses in each basic block represents its execution frequency. Variable `v1` is live in the basic blocks **A** and **B**. Since the call in basic block **B** is executed more frequently than the entry and exit basic blocks (**A** and **D**), it is more advantageous to map this variable onto a callee-saved register. Consequently, variable `v1` is saved and restored in respectively basic block **A** and **D**. For variable `v2`, it is more advantageous to assign it to a caller-saved register, because basic blocks **A** and **D** are more frequently executed than basic block **C**. The caller-saved store and restore code are inserted respectively before and after the call.

The TTA compiler back-end divides the registers of the target TTA equally into the two sets. To decide to which set a variable will be assigned, caller-saved and callee-saved costs are computed.

$$C_{\textit{caller-saved}}(v) = \sum_{call \in lr(v)} (2 \cdot L(\text{add}) + L(\text{ld}) + L(\text{st})) \cdot f(call) \tag{3.5}$$

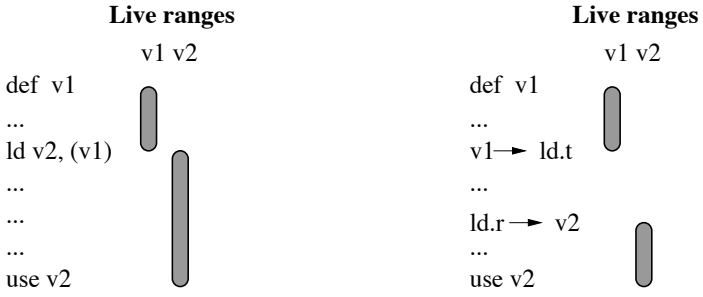
$$C_{\textit{callee-saved}}(v) = (2 \cdot L(\text{add}) + L(\text{ld}) + L(\text{st})) \cdot f(entry\ basic\ block) \tag{3.6}$$

where $lr(v)$ is the live range of variable v, $f(call)$ the execution frequency of the call and $f(entry\ basic\ block)$ the execution frequency of the first basic block of the procedure. The register allocator tries to map a variable in the set with

the lowest costs. When no registers of a set are available, a register from the other set is chosen.

3.3.4 TTA vs. OTA

TTAs have an advantage over most OTAs (Operation Triggered Architectures) in the context of register assignment. The live ranges of TTAs are finer grained. OTAs require that the register, which holds the result of an operation, is in use at the moment the operation starts executing, see Figure 3.12a [Cor98, page 268]. TTAs, however, do not have this limitation; a register is only in use at the moment it is defined, see Figure 3.12b. This may result in a lower register requirement.



a) Live range of an OTA. b) Live range of a TTA.

Figure 3.12: Live ranges of TTAs and OTAs.

3.4 Instruction Scheduling

The aim of instruction scheduling is to reorder operations and packing them into a minimum number of instructions, in order to reduce the execution time of an application, while preserving the semantics of the application and respecting the resource constraints. To ensure correct semantics of the produced schedule the dependences between operations should be respected. These dependences are given by the edges of the DDG. Furthermore, operations that are executed in parallel should not use the same hardware resources. A TTA instruction scheduler attempts to pack moves, instead of operations, into a minimum number of instructions. A TTA Instruction scheduler is more complex than an OTA instruction scheduler is. In this section, various instruction scheduling scopes and techniques in the context of TTAs are discussed. The presented algorithms are based on the work of Hoogerbrugge [Hoo96].

3.4.1 List Scheduling

Finding the optimal schedule is an NP-complete problem [GJ79]. A simple and efficient heuristic to schedule the code is *list scheduling*. It is the most popular technique used for instruction scheduling nowadays because it gives most of the time optimal results and has a short compilation time [BR91, BEH91b, Ell86, Fis81, HA99, HMC+93]. It comes as no surprise that the instruction scheduler in the TTA compiler back-end is also based on list scheduling. List scheduling repeatedly assigns an operation to an instruction without backtracking or lookahead. List scheduling schedules the operations in topological order. This order is determined by the DDG. An operation is scheduled when all its predecessors in the DDG within its scheduling scope have been scheduled. Such an operation is said to be ready and is a member of the $ready$ set. Selecting operations from the $ready$ set guarantees that scheduling will never block, because there are only upwards exposed constraints. There are two variants of list scheduling: instruction and operation-based list scheduling.

- *Instruction-based* list scheduling tries to place as many operations as possible in the current instruction, while respecting data dependence and resource constraints. When no more operations can be placed in the current instruction it proceeds to the next instruction.
- *Operation-based* list scheduling repeatedly selects a ready operation and places it in the first instruction where dependences and resources constraints are satisfied. In contrast with instruction-based list scheduling, this method does not create a schedule by filling one instruction at a time. Operation-based list scheduling is more general than instruction-based list scheduling because the number of operations in its ready set is larger: $ready(\textit{instruction-based}) \subseteq ready(\textit{operation-based})$. Consequently, operation-based list scheduling has a larger freedom in selecting operations for scheduling and potentially can achieve a higher performance.

Instruction-based list scheduling is more popular than operation-based list scheduling due to its lower engineering complexity. Unfortunately, instruction-based list scheduling is less suitable for TTAs because of the following three reasons[Hoo96]:

- Inefficient hardware usage. Scheduling the operand and trigger move individually, can result in a schedule in which they are scheduled far away from each other. In the interjacent instructions no other operations can use the operand register of the FU.
- Deadlocks. Scheduling moves individually can result in, for example, a situation where the trigger move of operation o_1 cannot be scheduled, because it depends on an operation o_2 that needs the same FU, and o_2 cannot be scheduled because operation o_1 has occupied the operand register.

- Pipelining problems. When VTL pipelined FUs are used, the results must be read in time before they are overwritten by successive operations scheduled on the same FU. When it is impossible to schedule the result move in time due to dependence or resource constraints, the operation must be unscheduled. This requires a backtracking scheduler, which increases the compilation time and the engineering complexity of the scheduler.

Because of the above mentioned problems, the TTA scheduler uses the operation-based list scheduling technique and schedules the moves of an operation in one indivisible step.

3.4.2 Resource Assignment

Resource parallelism in a processor exists in two forms [JW89]. The first is the ability of a processor to issue multiple instructions simultaneously. This is determined by the degree to which resources are duplicated in the processor. The second is the ability to overlap the execution of multiple operations, caused by the degree of pipelining of the FUs. Both forms of resource parallelism affect the extent to which a processor can exploit the ILP of an application.

Resource assignment assigns resources to operations. It is the responsibility of the scheduler to assign FUs to operations; buses and sockets to moves; and to decide whether an immediate is encoded in the source field of a move, or stored in an immediate field. In general, resource assignment in a TTA compiler is more complex than in an OTA compiler, because a TTA compiler has to assign more resources. Consequently, more resources have to be checked for conflicts.

The TTA compiler back-end uses a first-fit assignment algorithm to assign FUs, long immediates and sockets. The order of examination is determined by the order in which the resources are specified in the machine description file. Resources can have overlapping functionality; for instance, an FU can support a subset of operations of another FU in addition to its own operation set. To obtain the best results, resources with the most specialized functionality should be selected first. It is the responsibility of the designer to specify this order in the machine description file.

Move buses are assigned in a two step process [HC96]. A first-fit algorithm is used for finding a free move bus. When the interconnection network is fully connected and no move bus is found then it is guaranteed that no move bus is available. However, if the interconnection network is irregular, then reshuffling of the already made move bus allocation in the same instruction can lead to a valid allocation. A bipartite matching algorithm [DA93] is used for finding a valid move bus allocation. After scheduling, the actual move bus assignment is carried out.

Algorithm 3.2 SCHEDULEBASICBLOCK(b, E_{DDG})

$ready = \{o \in b \mid \neg\exists(o_i, o) \in E_{DDG}, o_i \in b\}$
$S = \emptyset$
WHILE $S \neq b$ **DO**
  o = SELECTOPERATION($ready$)
  IF ISCOPY(o) **THEN**
    SCHEDULECOPY(o)
  ELSE IF ISPROCEDURECALL(o) **THEN**
    SCHEDULEPROCEDURECALL(o)
  ELSE IF ISJUMP(o) **THEN**
    SCHEDULEJUMP(o)
  ELSE
    SCHEDULEOPERATION(o)
  ENDIF
  $S = S \cup \{o\}$
  $ready = \{o \mid o \in b - S \wedge \forall(o_i, o) \in E_{DDG}, o_i \in S\}$
ENDWHILE

3.4.3 Local Scheduling

There are several hierarchical levels or scopes at which instruction scheduling can be applied. The scheduling scope of a local or basic block scheduler consists of a single basic block. Scheduling decisions made in one basic block have no effect on scheduling decisions made in other basic blocks. Due to the limited size of basic blocks, typical 5 or 6 operations [JW89], the amount of ILP that can be exploited is modest. However, many scheduling techniques that exploit ILP in a larger scope use the principles of basic block scheduling.

Algorithm 3.2 shows the steps to create a schedule S of a basic block b. Operations for scheduling are selected from the set of ready operations. This process is repeated until all operations are scheduled. The order in which the operations are selected for scheduling has a large performance impact. Intuitively, operations along the critical path of the DDG should be scheduled first, since they are the primary bottleneck. The operations in the TTA compiler back-end are ordered with this observation in mind. The *slack based priority heuristic* is used for computing the priorities of the operations [Hoo96].

$$slack(o_i) = alap(o_i) - asap(o_i) \tag{3.7}$$

where $asap()$ is the *as-soon-as-possible* limit while respecting the data dependences. This limit represents the earliest instruction where operation o_i can be scheduled.

$$asap(o_i) = \begin{cases} \max\{asap(o_j) + delay(o_j, o_i)\} & : \textit{if } \exists(o_j, o_i) \in E_{DDG} \\ 0 & : \textit{otherwise} \end{cases} \tag{3.8}$$

where $delay(o_j, o_i)$ is the delay associated with the data dependence between the operations o_j and o_i. The function $alap()$ computes the *as-late-as-possible* limit; it represents the latest instruction where operation o_i can be scheduled without increasing the critical path length $L_{max}\,(b)$ of basic block b.

$$alap(o_i) = \begin{cases} \min\{alap(o_j) - delay(o_i, o_j)\} & : \textit{ if } \exists (o_i, o_j) \in E_{DDG} \\ L_{\max}\,(b) & : \textit{ otherwise} \end{cases} \tag{3.9}$$

The scheduler selects the operation from basic block b that minimizes $slack(o)$ and whose predecessors in the DDG are scheduled. The following priority function is used for ordering the operations:

$$priority_{slack}\,(o_i \in ready) = \left(1 - \frac{slack\,(o_i)}{L_{\max}\,(b)}\right) \tag{3.10}$$

Each time an operation is scheduled the priorities are recomputed.

The basic block scheduling algorithm distinguishes four types of operations: copies, procedure calls, jumps and data operations. Copy, jump and call operations are relatively easy to schedule. Each consists of a single move. Scheduling a data operation, like additions, subtractions, etc., is more difficult. These operations consist of multiple moves. For each move, resources have to be found. The moves of an operation o are scheduled with Algorithm 3.3. The earliest instruction in which an attempt is made to schedule a transport m_i of an operation o is computed with:

$$\text{EARLIESTINSN}(m_i) = \max_{m_j \in pred(m_i)} insn(m_j) + delay(m_j, m_i) \tag{3.11}$$

where $pred(m_i) = \{m_j \mid (m_j, m_i) \in E_{DDG}, m_j \in S\}$ and $insn(m_j)$ represents the instruction in which the transport m_j is scheduled. When $pred(m_i) = \emptyset$ then EARLIESTINSN(m_i) = 0. Scheduling attempts in earlier instructions will always fail because of the data dependence constraints. To obtain a compact schedule, the instruction counter i is incremented from the lower bound to the upper bound of the trigger move (see Algorithm 3.3). The upper bound is computed with:

$$\text{LATESTINSN}(m_t) = \text{LASTINSN}(S) + \text{NUMBEROFOPERANDMOVES}(o) + 1 \tag{3.12}$$

where LASTINSN(S) returns the number of instructions in the current schedule and NUMBEROFOPERANDS(o) gives the number of operand moves of operation o. The resources in the instructions between the last instruction of the current schedule and the upper bound are all free. Consequently, it is always possible to find free resources for the operand and trigger moves.

The first resource that is assigned to an operation is the FU. Assigning an FU to an operation differs from assigning move buses and sockets, in the sense that an FU is assigned to all moves of an operation, while move buses and sockets are assigned to each individual move. After an FU is found on which

Algorithm 3.3 SCHEDULEOPERATION(o)

```
FOR i = EARLIESTINSN(TRIGGER(o)) TO LATESTINSN(TRIGGER(o)) DO
  FOR EACH fu ∈ FU_set DO
    IF OPERATIONTYPE(o) ∈ OPERATIONSET(fu) THEN
      ASSIGNFU(o, fu)
      IF SCHEDULETRIGGERMOVE(m_t ∈ o, fu, i) THEN
        IF SCHEDULEOPERANDMOVES(o, fu, i) THEN
          IF SCHEDULERESULTMOVE(m_r ∈ o, fu, i) THEN
            return TRUE
          ENDIF
        ENDIF
      ENDIF
      RELEASERESOURCES(o)
    ENDIF
  ENDFOR
ENDFOR
return FALSE
```

Algorithm 3.4 SCHEDULETRIGGERMOVE(m_t, *fu*, i)

```
IF ISTRIGGERED(fu, i) THEN
  return FALSE
ELSE IF ¬ ASSIGNTRANSPORTRESOURCES(m_t, i) THEN
  return FALSE
ENDIF
return TRUE
```

the operation can be executed, the trigger, operand and result moves are scheduled. When scheduling of an operand and/or trigger move fails, the already assigned resources are released.

Algorithm 3.4 shows the steps necessary to schedule a trigger move m_t in instruction i. The algorithm first checks whether in instruction i another operation already performed a write to the *fu*'s trigger register. It is not allowed to write more than once in the same instruction to the same FU register. Scheduling of m_t succeeds when legal assignments can be found for the transport resources (sockets and move buses).

The operand moves[3] are scheduled using Algorithm 3.5. The first instruction, in which an attempt is made to schedule an operand move m_o, is equal to the instruction in which the trigger move is scheduled. If scheduling in this instruction fails, earlier instructions are tried. The earliest instruction in which the algorithm tries to schedule the operand move, is bounded by the

[3]Complex operations can have more than a single operand. For example `addmul` has two operands and one trigger: $(m_t + m_{o_1}) \cdot m_{o_2}$.

Algorithm 3.5 SCHEDULEOPERANDMOVES(*o*, *fu*, i)

```
FOR EACH m_o ∈ o DO
  FOR i' = i downto max (i − ot_freedom, EARLIESTINSN(m_o)) DO
    IF ISOPERANDREGISTEROCCUPIED(fu, i') THEN
      return FALSE
    ELSE IF ASSIGNTRANSPORTRESOURCES(m_o, i') THEN
      return TRUE
    ENDIF
  ENDFOR
ENDFOR
return FALSE
```

data dependence constraints of m_o and the parameter $ot_{freedom}$. This parameter ensures that the trigger and operand move are scheduled close to each other. This prevents inefficient hardware usage (see Section 3.4.1). A typical value for $ot_{freedom}$ is three[4]. Scheduling fails, if m_o overwrites an operand register, which is in use by another operation, or when no free transport resources (sockets and buses) can be found.

Algorithm 3.6 is used for scheduling the result moves. The first legal instruction in which the result move can be scheduled is equal to the instruction of the trigger move plus the latency of the operation. The upper bound is equal to the lower bound incremented with the constant $tr_{freedom}$. This constant prevents that the result move is scheduled too far from the trigger move. A typical value for $tr_{freedom}$ is three[5]. To prevent incorrect resource assignment three checks are required: (1) the trigger and result moves of all operations executing on *fu* have to be scheduled in FIFO order, (2) the number of operations in the pipeline may not exceed the capacity of the pipeline, and (3) on VTL pipelined FUs, collisions have to be prevented. The last check prevents that the contents of pipeline stages are unintentionally overwritten. If all these checks are passed, it is checked that no dependence constraints (output dependences) are violated: a result move cannot be scheduled earlier than its dependence constraints allow. The result move is scheduled successfully in instruction i', when legal assignments are found for the transport resources (sockets and move buses).

It should be stated that the presented algorithms give just a glimpse of the complexity of the issues involved for instruction scheduling for TTAs. Other issues are scheduling across procedure calls. All moves of an operation should be scheduled before or after the call. In addition, the exploitation of the TTA specific optimizations is done during scheduling.

[4] A value of zero would restricts the operand-trigger scheduling freedom completely. Experiments indicate that this may result in a performance loss of 7% [Hoo96].

[5] A value of zero would restricts the trigger-result scheduling freedom completely. Experiments indicate that this may result in a performance loss of 3% [Hoo96].

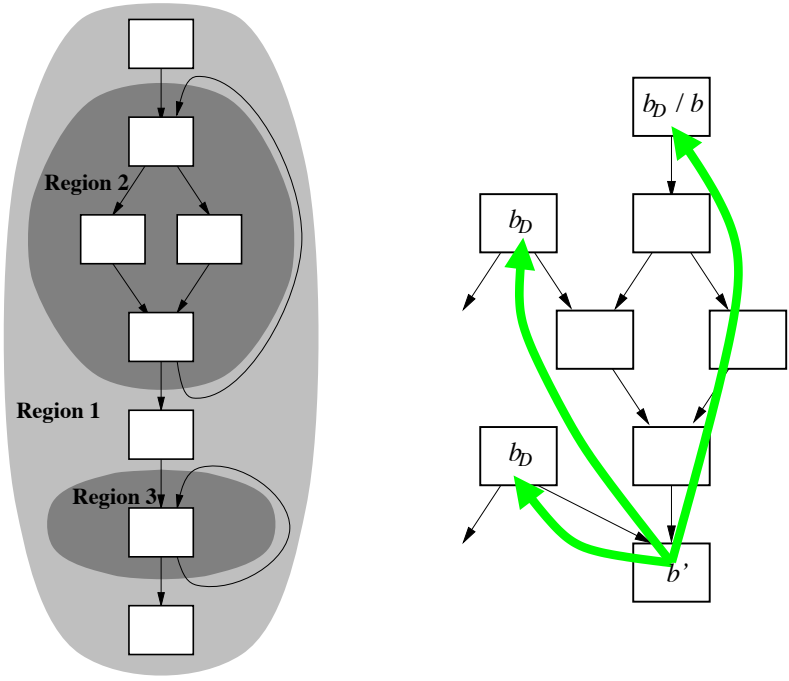
Algorithm 3.6 SCHEDULERESULTMOVE(m_r, *fu*, i)

```
FOR i' = i + LATENCY(fu) to i + LATENCY(fu) + tr_freedom DO
  IF NOTINFIFOORDER(fu, i') THEN
    return FALSE
  ELSE IF PIPELINEOVERFLOW(fu, i') THEN
    return FALSE
  ELSE IF ISVTLPIPELINE(fu) ∧ COLLISION(fu, i') THEN
    return FALSE
  ELSE IF i' < EARLIESTINSN(m_r) THEN
    continue
  ELSE IF ASSIGNTRANSPORTRESOURCES(m_r, i') THEN
    return TRUE
  ENDIF
ENDFOR
return FALSE
```

3.4.4 Global Scheduling

As already stated in the previous section, the size of a basic block is limited: typical 5 or 6 operations for non-numeric code. As a result, the amount of ILP that can be exploited by a basic block scheduler is limited. To increase the amount of exploitable ILP the scheduling scope should be larger than a single basic block. Scheduling scopes that exploit ILP across multiple basic blocks are called *extended basic block schedulers* or *global schedulers*. The scheduling scope of these schedulers consist of an acyclic CFG. The inter basic block ILP is exploited by moving operations between basic blocks belonging to the same scheduling scope. This may require speculative execution and code duplication. In literature various extended basic block scheduling scopes are introduced [BR91, Fis81, HMC+93, MLC+92, Mah96]. *Regions* [BR91] should potentially give, due to their generality, the best performance. Regions correspond to the body of a natural loop. Note that this body may contain arbitrary nested conditional statements. Figure 3.13a shows the region hierarchy of a program. The region scheduler used in the TTA compiler back-end is inspired on the work of Bernstein [BR91] and has extensions for multi-way branching and predicated execution.

Basic blocks of a region are scheduled in topological order; a basic block is scheduled after all its predecessors are scheduled. Region scheduling enlarges the exploitable ILP by moving operations over basic block boundaries. This is illustrated in Figure 3.13b. First, all operations of destination basic block b are scheduled using the basic block scheduler. Afterwards, the remaining operations in the sequential code of the same region are examined, to find operations from other basic blocks that can be scheduled in b, see Algorithm 3.7. The process of scheduling an operation into another basic block is called *importing*



a) Nesting of regions. b) Code motion in a region.

Figure 3.13: Regions.

operations. When an operation o is selected for importing, it is removed from its source basic block b' and added to the destination basic block b. To ensure correct semantics, code duplication might be necessary. Consider Figure 3.13b, moving an operation from basic b' to basic block b requires the insertion of duplicates in the basic blocks b_D in order to preserve the semantics of the program. Duplication of an operation is required when basic block b does not dominate b'. The algorithm to compute the set of duplication basic blocks D can be found in [Hoo96]; D includes the destination basic block b. To simplify scheduling, no control paths are allowed between the elements of D. This is known as the single copy on a path rule (SCP) described in [BCK91].

Algorithm 3.8 is used for importing an operation o in a basic block b. The function BB(o) returns the (source) basic block of operation o. Note that some of the duplication basic blocks D might be scheduled already. In the present implementation, the imported operations are not permitted to enlarge these basic blocks. Importing also fails when in one of the scheduled duplication basic blocks insufficient resources are available.

The code motion from basic block b' to b in Figure 3.13b is always profitable because all execution paths starting at b go through b'. This is not the case for the other two duplication basic blocks. If the outcome of the branch in one of

Algorithm 3.7 SCHEDULEREGION($Region$, E_{DDG})

$is\_scheduled = \emptyset$
FOR EACH $b \in Region$ IN TOPOLOGICAL ORDER **DO**
  SCHEDULEBASICBLOCK(b, E_{DDG})
  $is\_scheduled = is\_scheduled \cup \{b\}$
  WHILE NOT EACH REACHABLE OPERATION o TRIED **DO**
    TRYTOIMPORTOPERATION(b, o)
  ENDWHILE
ENDFOR

Algorithm 3.8 TRYTOIMPORTOPERATION(b, o)

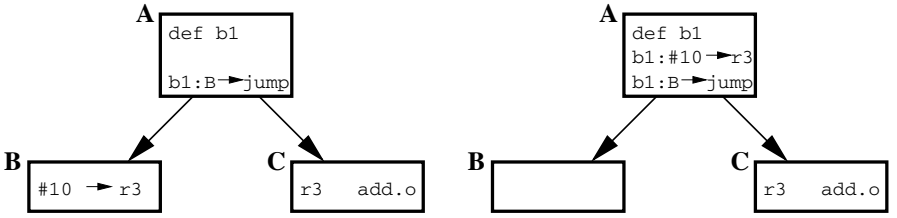
$b' =$ BB(o)
$D =$ COMPUTEDUPLICATIONSET(b, b')
FOR EACH $b'' \in D$ **DO**
  IF $b'' \in is\_scheduled$ **THEN**
    IF $\neg$ TRYTOSCHEDULEOPERATION(b'',o) **THEN**
      RELEASERESOURCES(D, o)
      return
    ENDIF
  ENDIF
  $b'' = b'' \cup \{o\}$
ENDFOR
$b' = b' - \{o\}$

these duplication basic blocks is not in the direction of basic block b', then the operation imported from basic block b' is executed, but its result is never used. Code motions into basic blocks containing jumps that might not branch in the direction of b', are said to be *speculative*.

Some operations are not allowed to be speculatively executed. Speculatively imported operations that produce exceptions, overwrite registers or overwrite memory locations, may change the state of a program when the outcome of the branch is not in the direction of b'. This may lead to incorrect program execution. To eliminate these restrictions operations can be guarded[6], see Figure 3.14. The copy operation is guarded by a guard expression, that is computed by combining the guards of the branches over which the operation is imported. When the operation was not guarded and the jump branched to basic block **C**, the imported operation would unintentionally overwrite register `r3` and would change the outcome of the addition. This is also known as *off-liveness*.

Instead of moving whole operations across basic block boundaries, it can also

[6]Operations that produce exceptions are never speculatively executed.



a) Original code fragment. b) Code fragment after importing.

Figure 3.14: Speculative code motion with guarding.

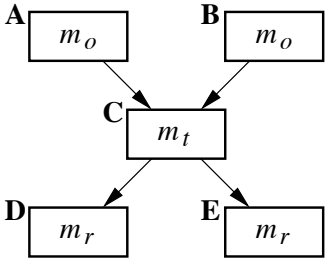


Figure 3.15: Scheduling over basic block boundaries.

be advantageous to import individual moves. The TTA region scheduler tries to schedule individual moves across basic block boundaries in the following situations, see Figure 3.15:

- When the trigger move of an operation is scheduled in basic block **C** and the operand move cannot be placed in the same or earlier instruction of **C**, then the scheduler will try to schedule the operand move in the predecessor basic blocks **A** and **B**.
- When the result move cannot be placed in an instruction in **C**, the scheduler tries to place the result move in the successor basic blocks **D** and **E**.

To prevent inefficient hardware usage, due to a large number of instructions between the moves of an operation, individual moves are only imported in direct successor or predecessor basic blocks.

The performance of an extended basic block scheduler depends on the order in which ready operations are tried for importing. The priority function used for the TTA region scheduler [Hoo96] is given by:

$$priority\,(o) = Pr\,(b'|b) \cdot \left(1 - \frac{slack\,(o, b')}{L_{max}\,(b')}\right) \tag{3.13}$$

where o is the candidate operation for the code motion from basic block b' to basic block b and $Pr\,(b'|b)$ is the probability that b' will be executed after b. A large probability makes it very likely that the imported operation will indeed be executed and hence the used resources are spend well. The parameter $slack\,(o, b')$

gives the slack or criticality of operation o in basic block b'. This parameter is normalized with $L_{max}(b')$, the critical path length of b', to prevent that operations from small basic blocks are prioritized over operations from larger basic blocks. The operation with the highest priority is selected for importing. This process is repeated until all ready operations have been tried.

3.4.5 Software Pipelining

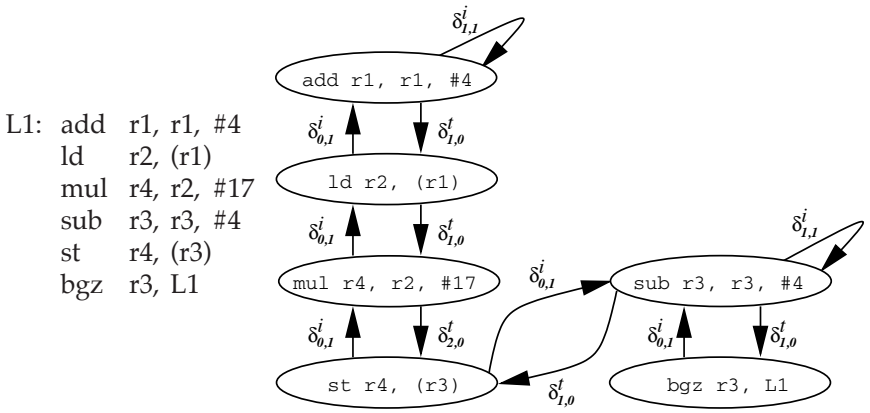
As already mentioned in Section 3.2.5, loop unrolling enables the exploitation of ILP of successive loop iterations. However, replicating the loop body increases the code size. Furthermore, prior to scheduling, one has to decide how many times to unroll the loop. *Software pipelining* is a technique that potentially achieves the same performance as infinite loop unrolling with only a modest code size expansion.

Software pipelining has received widespread attention in academic and industrial research [Lam88, Rau94, LVAG95]. This scheduling technique exploits ILP of loops by overlapping the execution of successive iterations. During the execution of a non-pipelined loop, the first iteration is started and executed to its completion. The second iteration is then initiated and executed until completion, etc. In contrast, for a software pipelined loop, the second iteration of the original loop is allowed to start before the first iteration is completed. The interval at which iterations of the software pipelined loop are started is called the *initiation interval* (II), which is expressed in the number of cycles. The goal of software pipelining is to find a schedule with the shortest possible initiation interval. A number of methods on construction of software pipelined loops were published, the most widely used is *Iterative Modulo scheduling* [Rau94]. The algorithm chosen for the research presented in this thesis is also based on this method.

Iterative modulo scheduling first computes a lower bound on the II called the *minimum initiation interval* (MII). Then, it tries to schedule the loop in MII instructions. When scheduling fails the II is increased until a valid schedule is constructed that fits. In order to reduce the number of iterations required to generate a valid schedule, the MII should be estimated as precisely as possible. Resource and dependence constraints are used for this estimation. For more information on computing the MII the reader is referred to the literature [Rau94].

Basic block and region scheduling are hindered by resource and dependence constraints between operations in the same iteration. Software pipelining also has to respect dependences between operations of different iterations. This is accomplished by adding to the DDG so-called *inter-iteration data dependences*, denoted as $o_i \ \delta^i_{delay,distance} \ o_j$. This data dependence edge states that o_j should be executed at least $delay$ cycles after o_i in the $distance^{th}$ previous iteration.

Figure 3.16a shows the RISC style code of the loop body that executes the



a) Example loop. b) DDG of the loop.

```
                        Prologue
    add r1, r1, #4
    ld  r2, (r1)
    add r1, r1, #4   mul r4, r2, #17
    ld  r2, (r1)     sub r3, r3, #4
                        Kernel
L1: add r1, r1, #4   mul r4, r2, #17  st  r4, (r3)
    ld  r2, (r1)     sub r3, r3, #4   bgz r3, L1
                        Epilogue
                     mul r4, r2, #17  st  r4, (r3)
                     sub r3, r3, #4
                                      st  r4, (r3)
```

c) Software pipeline with $II = 2$.

Figure 3.16: Example software pipelining.

computation `b[n..1] = 17 * a[1..n]`. For reasons of clarity, OTA code is used in the example instead of TTA code. The DDG with inter-iteration dependences is given in Figure 3.16b. Scheduling the loop with a basic block scheduler results in 5 cycles per loop iteration, assuming a latency of two for the multiplier, a single cycle latency for all other operations and infinite resources. Its software pipelined counterpart in Figure 3.16c starts each second cycle a new iteration.

The schedule produced by a software pipelining algorithm consists of three different phases: the *prologue*, the *kernel* and the *epilogue*. The pipeline is started by the prologue, then the kernel is executed (the loop-body of the schedule), and finally the epilogue drains the pipeline. For high iteration counts, the kernel mainly determines the total execution time.

The software pipelining algorithm (see the Algorithms 3.9 and 3.10) is based

Algorithm 3.9 SOFTWAREPIPELINING(b)

II = COMPUTEMII(b)
$budget = budget\_ratio \cdot |b|$
WHILE $\neg$ ITERATIVEMODULOSCHEDULING(b, II, $budget$, *Huff*) $\wedge$
$\neg$ ITERATIVEMODULOSCHEDULING(b, II, $budget$, *Rau*) **DO**
$II = II + 1$
ENDWHILE

on the algorithm for operation-based list scheduling for basic blocks. The main differences are:

- In contrast to basic block scheduling, resource conflicts not only can arise in the same instruction i, but also in all instructions $i + II \cdot k \ \forall k > 0$. Therefore, when an operation uses a resource in instruction i, the state of this resource in instruction $i \ mod \ II$ is updated. Checking whether a resource can be used in instruction i, means checking the availability of this resource in instruction $i \ mod \ II$.
- In local and global scheduling, the first instruction in which an attempt is made to schedule an operation o_i is computed with Equation 3.11. This computation expects that all predecessors are already scheduled. However, because of the recurrences in the DDG it is not always possible to select an operation for scheduling whose predecessors are all scheduled. Furthermore, also the inter-iteration dependences must be included.

  $$EarliestInsn(o_i) = \max_{o_j \in pred^*(o_i)} \{insn(o_j) + delay(o_j, o_i) - II \cdot distance(o_j, o_i)\}$$

  where $pred^*(o_i)$ is the set of scheduled predecessors of operation o_i and $\max(\emptyset) = 0$. The latest instruction in which an operation o_i is tried is equal to $EarliestInsn(o_i) + II - 1$. Searching beyond this boundary is useless because if there is a resource conflict in instruction i, then there is also a resource conflict in instruction $i + k \cdot II$.
- Because an operation can be scheduled before all its predecessors are scheduled, a dependence constraint can be violated. When an operation o_i is scheduled, the following condition is checked:

  $$insn(o_j) < insn(o_i) + delay(o_i, o_j) - II \cdot distance(o_i, o_j) \ \ \forall o_j \in succ(o_i)$$

  where $succ(o_i) = \{o_j \mid (o_i, o_j) \in E_{DDG}, o_j \in S\}$. When this expression evaluates to true, iterative modulo scheduling corrects the partial schedule by unscheduling all operations that conflict with operation o_i.
- As already observed, it is not always possible to generate a correct schedule in II instructions. To detect the inability to schedule the loop within the given II a scheduling budget is provided. This budget equals $budget\_ratio \cdot |b|$ where $|b|$ is the number of operations in the software

Algorithm 3.10 ITERATIVEMODULOSCHEDULING(b, II, $budget$, $heuristic$)

```
S = ∅
WHILE S ≠ b ∧ budget > 0 DO
  o = SELECTOPERATION(b - S, heuristic)
  IF ISCOPY(o) THEN
    IF ¬ SCHEDULECOPY(o) THEN
      return FALSE
    ENDIF
  ELSE IF ISJUMP(o) THEN
    IF ¬ SCHEDULEJUMP(o) THEN
      return FALSE
    ENDIF
  ELSE
    IF ¬ SCHEDULEOPERATION(o) THEN
      return FALSE
    ENDIF
  ENDIF
  S = S ∪ {o}
  budget = budget - 1
  IF ¬ DEPENDENCESCORRECT(S, o) THEN
    UNSCHEDULEALLCONFLICTINGOPERATIONS(S, o)
  ENDIF
ENDWHILE
return TRUE
```

pipelined loop and $budget\_ratio$ an adjustable parameter (a typical value is 4.5). The scheduling budget is decremented each time an operation is scheduled. Scheduling fails when the budget becomes negative. The larger the value of $budget\_ratio$, the harder the scheduler tries to find a valid schedule.

- The order in which the operations are selected for scheduling influences the efficiency of the generated schedule. Because operations from different iterations are executed at the same time, Huff [Huf93] modified the slack based priority heuristic of Equation 3.10 by replacing the delay of a DDG edge by the length of that edge ($delay(o_i, o_j) - II \cdot distance(o_i, o_j)$). Another popular priority heuristic is the height-based priority function proposed by Rau [Rau96]. This heuristic gives a higher priority to operations with a large height. The height of an operation is defined as the length of the longest path in the DDG from the operation to a stop pseudo operation that is dependent on all operations in the loop. Hoogerbrugge [Hoo96] evaluated both priority functions. Neither appeared to be clearly better than the other. As proposed by Hoogerbrugge both

heuristics are tried in order to generate a valid schedule in II instructions.

A drawback of modulo scheduling is that it can only handle single basic block loops. To overcome this problem if-conversion [WHB92] is used. This method combines the branches of an if-then-else construction into a single basic block with the use of predicates or guards. This method is also applied in the TTA compiler back-end. Software pipelining can only be performed on the inner most loops, therefore the remaining parts of the code are handled by the region scheduler.

Evaluation Methodology

4

To evaluate the quality of the introduced algorithms and compiler strategies an experimental framework is defined. This experimental framework consists of 30 benchmark applications, two TTA processors and a measurement methodology. Measurements play an important role in evaluating compiler techniques. Analysis of the results may lead to various improvements and in a better understanding of the proposed methods.

The application benchmark suite is described in Section 4.1. This suite consists of workstation-type applications, benchmarks from the SPECint95 benchmark suite [Spe96], benchmarks from the MediaBench suite [LPMS97], and DSP (Digital Signal Processing) benchmarks. In Section 4.2, the TTA processor configurations used in the experiments are described. To give an impression of the quality of the compiler in combination with the selected TTA processors, some performance metrics are provided. Section 4.3 evaluates the three scheduling scopes as discussed in Section 3.4. In Section 4.4, the achieved amount of instruction-level parallelism (ILP) is given when the benchmarks are compiled with the selected TTA processors.

4.1 Benchmark Suite

Benchmark applications are used for the evaluation of the developed algorithms. The benchmarks are selected from various application areas in order to prevent that the developed algorithms are tailored towards a specific application or application area. Only real-world applications are taken, synthetic benchmarks such as Dhrystone or the Livermore loops are not considered. The benchmark suite consists of:

- Workstation-type benchmarks. Most of these benchmarks will never be considered to be mapped onto an ASP (Application Specific Processor). These benchmarks give a good indication of the quality of the compiler algorithms when used in combination with general-purpose processors.
- Benchmarks of the SPECint95 suite [Spe96]. The SPECint95 suite is an internationally recognized benchmark suite that represents a typical workload. The amount of ILP is expected to be varying. For example, *132.ijpeg* is expected to have a high degree of ILP because the algorithms used in this benchmark have a parallel nature. The benchmarks *147.vortex* and *099.go* are likely to have a low degree of ILP, because usually a database program respectively a game are dominated by control intensive code[1].
- DSP benchmarks obtained from [Emb95]. These benchmarks contain small loops, which are executed frequently. The algorithms used within these benchmarks are representative algorithms that are especially suitable for implementation within an ASP. A high degree of ILP is expected.
- Benchmarks from the MediaBench suite [LPMS97]. These applications apply DSP like algorithms. In addition, they also contain control intensive code. These applications are candidates for implementation in ASPs.

The benchmarks and their characteristics are listed in Table 4.1. The table lists for each benchmark the following characteristics: (1) a short description of the benchmark, (2) the number of static operations including the library code, which gives an indication of the code size, and (3) the number of dynamic (executed) operations. This last number highly depends on the input data sets taken for each benchmark.

Code efficiency is measured by the execution time of an application. When averaging the results of the measurements, it is assumed that each benchmark is equally important (independent of the static or dynamic code size of the benchmark). The presented results in the remainder of this dissertation are the (unweighted) arithmetic means of the individual measurements.

4.2 TTA Processor Suite

In this section, the TTA processors used for evaluating the developed methods are described. The TTA processor selection method is described in Section 4.2.1. The selected TTA processors are described in detail in Section 4.2.2.

4.2.1 Space Walking

Designing an Application Specific Processor (ASP) consists of finding a proper set of resources for the given application or application domain. The ASP de-

[1] A game like Go usually contains more task parallelism than instruction-level parallelism.

Table 4.1: Benchmark characteristics.

Benchmark	**Description**	**#static oper.**	**#dyn. oper.**
Workstation-type			
a68	68K assembler	19646	2805K
bison	Parser generator	18962	4902K
cpp	C preprocessor	15833	1960K
crypt	Encryption	4253	5875K
diff	File compare	21802	29M
expand	Tab expansion	4895	29M
flex	Scanner generator	19567	12M
gawk	Language interpreter	36157	42M
gzip	File compression	14539	108M
od	Octal dump	7315	21M
sed	Stream editor	17532	46M
sort	Sort lines	7908	81M
uniq	Report repeated lines	5368	27M
virtex	Text formatting	41789	50M
wc	Word count	4481	7192K
SPECint95			
099.go	Plays the game of Go	133K	236G
124.m88ksim	Microprocessor simulator	29K	73G
129.compress	Data compression	7051	45G
132.ijpeg	JPEG encoder	39K	92G
147.vortex	Object-oriented database	122K	72G
DSP-type			
instf	Frequency tracking	1978	3140K
mulaw	Speech compression	1397	330K
radproc	Doppler radar processing	1903	29M
rfast	Fast convolution using FFT	1946	3098K
rtpse	Spectrum analysis	1936	2090K
MediaBench			
djpeg	JPEG decoder	26K	5568K
rawcaudio	Audio encoder	3486	8144K
mpeg2decode	MPEG2 decoder	13336	168M
mpeg2encode	MPEG2 encoder	20887	1662M
unepic	Wavelet decoder	9538	7484K

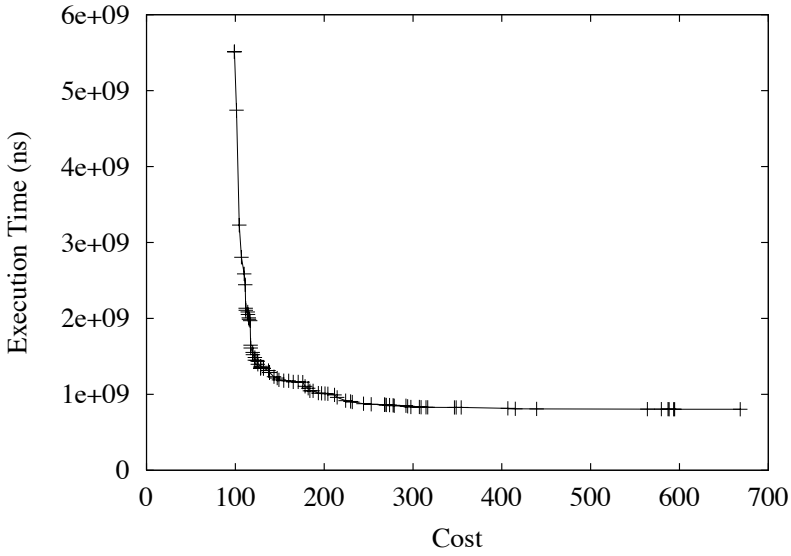
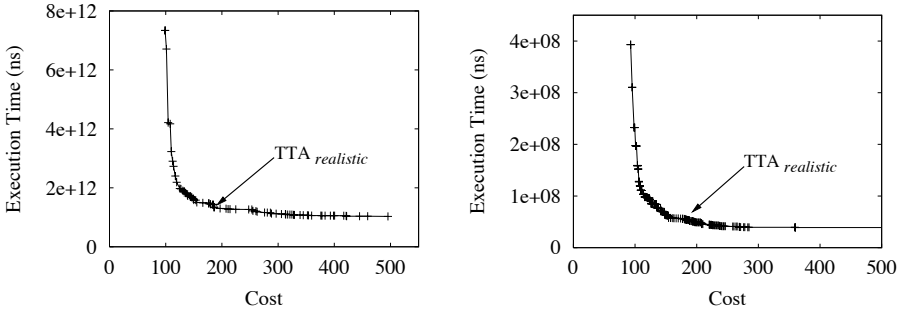


Figure 4.1: Curve generated by the synthesis tool for benchmark *sort*.

sign space is very large. Manual exploration is tedious and error prone. Methods to automate the design space exploitation are described in [FFD96, HC95]. In [FFD96], a system, which automatically designs VLIW architectures for a given application, is described. The design methodology as proposed in [HC95], designs ASPs based on TTAs. This synthesis tool is used for selecting the TTA configurations for this thesis. A popular term to refer to these techniques is *space walking*.

Selecting a proper TTA configuration for a given application is a trade-off between parameters such as performance, chip area, power consumption, code size, etc. These objectives are in conflict and the relative importance depends on the application. It is hard to select a TTA processor whose parameters are close to the desired requirements in a single step. The used synthesis tool performs a quantitative analysis of many design points. The result of this tool is shown in Figure 4.1 when applied to the *sort* application. Each point on the cost-performance curve is a 2-tuple $(t_{exec}, cost)$ and corresponds to a particular TTA processor. The execution time t_{exec} is the estimated time in ns to run the application and is the product of the cycle count and the cycle time. The compiler determines the cycle count. The cycle time is computed using a cycle time model. The parameter $cost$ represents the realization cost. This parameter is expressed in units of a 32-bit integer function unit[2]. The design points on the curve are called Pareto points [dM94]. A configuration is a Pareto point

[2]For a more detailed description of the hardware cost model and the cycle time model the reader is referred to [CH95, Hoo96]. In [Arn01] a new more accurate model is presented.



a) Space walking curve for *132.ijpeg*. b) Space walking curve for *crypt*.

Figure 4.2: Space walking curves.

if it is realizable and there are no other realizable configurations that are both faster and cheaper. In other words, the curve gives the lowest cost for a given performance or reverse, gives the best performance for a given cost.

4.2.2 Selected TTA Processors

The synthesis tool, as described in the previous section, is applied to select a realistic TTA processor ($TTA_{realistic}$) for the experiments. The space walking curves of various benchmarks are used to make this selection; the results of two of them are shown in Figure 4.2. The selected TTA processor should combine good performance with reasonable cost. Processors that comply with this requirement can be found in the knee of the curves as indicate by the arrows in Figure 4.2. Besides a realistic TTA processor, a second TTA processor

Table 4.2: Function units characteristics.

FU name	Latency	Pipelined	Operations
load/store FU	2	y	`ld`, `ldb`, `ldh`, `ldd`, `lds`, `st` ,`stb`, `sth`, `std`, `sts`
integer FU	1	-	`add`, `sub`, `eq`, `gt`, `gtu`, `shl`, `shr`, `shru`, `and`, `ior`, `xor`, `sxbh`, `sxbw`, `sxhw`
integer multiply FU	3	y	`mul`
integer divide FU	8	n	`div`, `divu`, `mod`, `modu`
floating-point FU	3	y	`addf`, `subf`, `negf`,`mulf`, `eqf`, `gtf`, `f2i`, `f2u`, `i2f`, `u2f`, `divf`

Table 4.3: Supported guard expressions; $\mathtt{b}_x$ and $\mathtt{b}_x$ are Boolean registers.

Simple expressions	$\mathtt{b}_x$, $\mathtt{b}_y$, $\mathtt{!b}_x$, $\mathtt{!b}_y$
And expressions	$\mathtt{b}_x \cdot \mathtt{b}_y$, $\mathtt{!b}_x \cdot \mathtt{b}_y$, $\mathtt{b}_x \cdot \mathtt{!b}_y$, $\mathtt{!b}_x \cdot \mathtt{!b}_y$
Or expressions	$\mathtt{b}_x + \mathtt{b}_y$, $\mathtt{!b}_x + \mathtt{b}_y$, $\mathtt{b}_x + \mathtt{!b}_y$, $\mathtt{!b}_x + \mathtt{!b}_y$

Table 4.4: Benchmark TTA configurations.

Resource		Configuration	
		$\text{TTA}_{realistic}$	TTA_{ideal}
# move buses		8	64
# load/store FUs		2	16
# integer FUs		3	24
# integer multiply FUs		1	8
# integer divide FUs		1	8
# floating-point FUs		1	8
immediate	# short(8-bits)	8	64
	# long (32 bits)	2	32
integer RF	# registers	n	n
	# read ports	4	32
	# write ports	4	32
floating-point RF	# registers	48	512
	# read ports	2	32
	# write ports	1	32
Boolean RF	# registers	4	32
	# write ports	2	32

is added to the processor benchmark suite. This second TTA processor, named TTA_{ideal}, has many resources of each type. Because compilation for this processor is hardly hindered by resource constraints, it is well suited to evaluate the potential performance of new algorithms. Note that the TTA_{ideal} does not correspond to TTA configurations at the uttermost right of the space walking curves. The enormous amount of connections to the move buses results in a large cycle time and hence the execution time increases significantly. In this respect, the TTA_{ideal} configuration would be in the upper right corner. In the remainder of this thesis, the impact of the cycle time is ignored (unless stated explicitly) because we are mainly interested in the quality of the generated code.

The function units (FUs) characteristics of the $\text{TTA}_{realistic}$ and TTA_{ideal} processors are listed in Table 4.2. The pipelined FUs use the virtual time latching (VTL) pipeline discipline as discussed in Section 2.2.2. Both TTA processors support guarding. Each move bus is guarded by guard expressions. When this expression evaluates to true the associated transport is executed, otherwise the

transport is squashed. Hoogerbrugge [Hoo96] claims that a guard expression size of more than two does not give much performance gain. Therefore, in our experiments we use an expression size of two. The available guard expressions are listed in Table 4.3.

The TTA templates assume a jump latency of two cycles. This results in one delay slot. The interconnection network is fully connected. The memory system is assumed to be perfect. Cache or TLB misses are not taken into consideration. Note that many embedded processors have local memories (instead of caches) and no virtual memory. The parameters of the two TTA processors are listed in Table 4.4. The number of read ports of the Boolean register file is not listed because Booleans are read implicitly by guards. The number of integer registers n is varied during the experiments between 10 and 512.

4.3 Scheduling Scopes

In Section 3.4, three scheduling scopes were discussed: (1) basic block scheduling, (2) region scheduling and (3) software pipelining. These scheduling scopes play an important role in this dissertation. In this section, the effect of the scheduling scope on the performance is measured. Figure 4.3 gives the speedup relative to basic block scheduling when using the TTA_{ideal} template with 512 registers. As can be seen, region scheduling results in a large improvement, on average 135%. These results clearly demonstrate that exploiting ILP across basic block boundaries is beneficial.

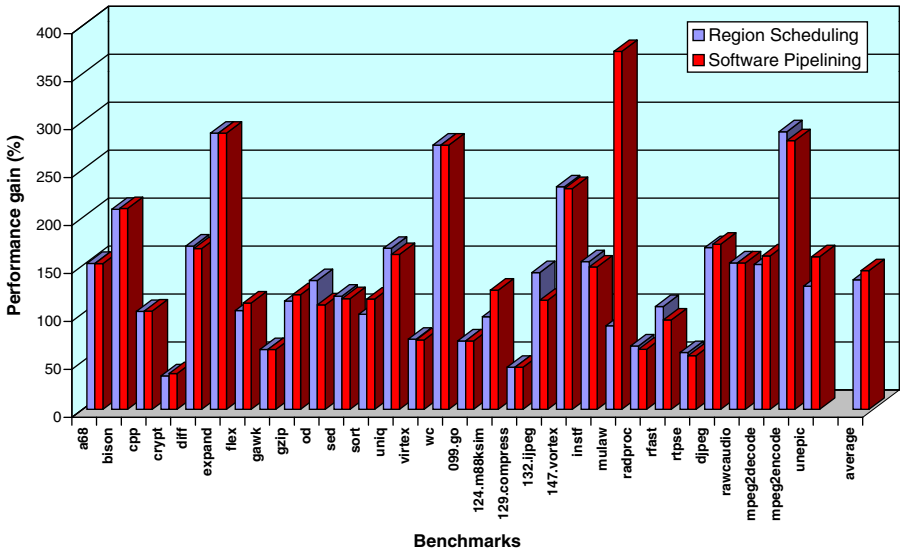


Figure 4.3: Performance gains of region scheduling and software pipelining relative to basic block scheduling.

Table 4.5: The software pipeline ratio.

Benchmark	Software pipeline ratio
crypt	0.840
flex	0.148
gzip	0.170
od	0.437
sort	0.172
virtex	0.109
124.m88ksim	0.207
132.ijpeg	0.506
instf	0.735
mulaw	0.995
radproc	0.246
rfast	0.583
rtpse	0.147
djpeg	0.425
mpeg2decode	0.588
mpeg2encode	0.193
unepic	0.229

The average performance gain of software pipelining is even larger, 145%. However, this gain difference is mainly caused by the benchmark *mulaw*, which resulted in a speedup of 373%. The execution time of this benchmark is dominated by a single loop, which is very suitable for software pipelining. When the benchmark *mulaw* is ignored, software pipelining performs only slightly better than region scheduling. In [Hoo96], various reasons are mentioned to explain this modest improvement. The two most important are: (1) without software pipelining, the loops are unrolled which is already quite effective and (2) only a fraction of the loops can be software pipelined[3]. Table 4.5 shows the *software pipeline ratio* for various benchmarks. The software pipeline ratio represents the fraction of the execution time spend in software pipelined loops. Only benchmarks with a software pipeline ratio higher than 10% are listed. In addition, a third reason for the modest improvement of software pipelining is identified. Software pipelining generates prologue and epilogue basic blocks. In the current implementation local scheduling is applied to these basic blocks. Better results can be achieved when this code is scheduled together with the code that surrounds the original loop. To achieve high performance a best-of-both-worlds strategy seems profitable. Such a strategy would generate per loop two schedules: one using region scheduling and one using software pipelining. The one with the highest performance should be incorporated in the total schedule.

[3] As can be observed many benchmarks give the same results for region scheduling and software pipelining which indicates that no loops, or only a small fraction, could be software pipelined.

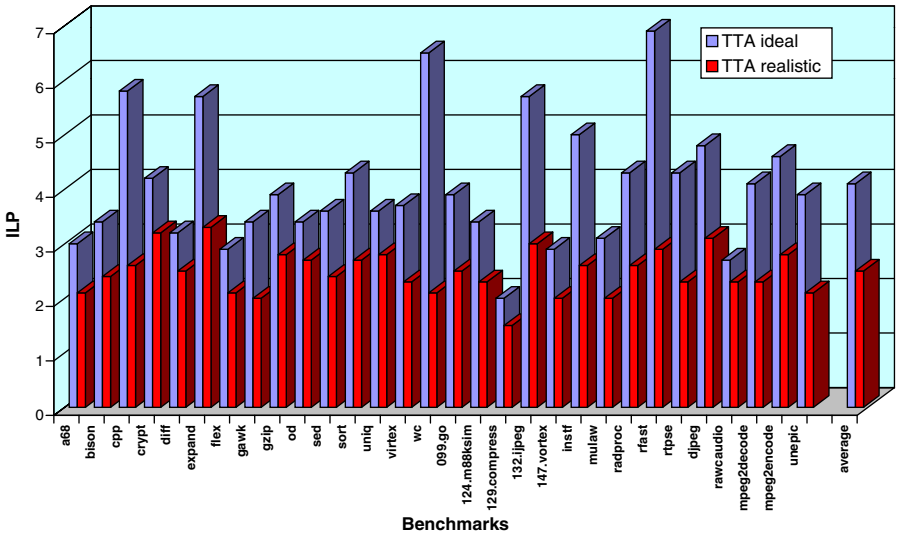


Figure 4.4: Exploited ILP for the TTA_{ideal} and $\text{TTA}_{realistic}$ processors.

4.4 Exploitable ILP

In the previous section, we saw that region scheduling and software pipelining generate code with a much higher performance than basic block scheduling. Consequently, also the amount of exploited ILP is increased. The practically exploitable ILP of an application depends on the nature of the application (control-intensive, DSP or multimedia) and the used processor. Studies [JW89, Wal91, LW92, TGH92, LW97] to measure the maximum available ILP assume the presence of infinite processor resources and perfect predictors to predict the behavior of a program. In this section, the exploitable ILP of the benchmark applications is measured using both TTA processors and the region scheduler. To allow the exploitation of large amounts of ILP, no register assignment is carried out. Consequently, the code does not contain spill and state preserving code. Although the TTA_{ideal} processor is not realistic for practical use, its (hardware) ability to exploit large amounts of ILP gives an indication of the performance of the TTA compiler. The exploited ILP varies between 1.5 and 6.9. Figure 4.4 gives the results for the individual benchmarks. As can be seen the DSP-type benchmarks and benchmarks from the MediaBench suite have a higher degree of ILP than the more control intensive Workstation-type applications. Because the TTA_{ideal} processor contains more resources, the exploited ILP is larger than the exploited ILP for the $\text{TTA}_{realistic}$ processor.

The Phase Ordering Problem

5

Modern optimizing compilers consist of several optimization phases. An important research topic in compiler design is to find the optimal phase ordering. In this thesis, we focus on the phase ordering of the two most important phases in ILP compilers: register assignment and instruction scheduling [HP90]. Both phases have received widespread attention in academic and industrial research [ASU85, BR91, Bri92, CH90, CAC+81, CH95, Ell86, Fis81, GL95, GS90, HHR95, Hoo96, Lam98, ME92, NP95, Pin93, Rau94, WM95]. The interaction between these two phases is becoming increasingly important, since the number of simultaneously executed operations increases due to advances in silicon and compiler technology. Executing more operations simultaneously results in a higher register pressure.

An important question to answer is when, during compilation, should register assignment take place. In one sense, one would like register assignment to be done very late in the compilation process. This approach maintains the myth of unlimited registers until after traditional optimizations, such as common subexpression elimination, copy propagation, and dead code removal. These optimizations increase or reduce the number of required registers by creating or removing variables, or by changing the live range of variables. Since register assignment is not yet applied, it is valid to create variables and to alter their live ranges. If registers are assigned before one or more of these optimizations, assignment and spilling decisions are based on a poor estimate of the register usage.

So far, the timing of register assignment with respect to traditional compiler optimizations is discussed. How does inclusion of an instruction scheduling phase affect the optimal placement of register assignment? While scheduling

itself will not create variables, it will most certainly alter the live ranges of variables by changing the relative order of operations. Applying register assignment first, limits the instruction scheduler's ability to reorder operations. Applying scheduling first, most likely results in schedules that require more registers than available. The interaction between register assignment and instruction scheduling has its impact on the produced code; decisions made by one phase can have negative effects on the other. The order, in which these two phases should be applied, is a point of dispute.

This chapter gives an overview of several strategies towards the phase ordering of register assignment and instruction scheduling. Section 5.1 describes and evaluates methods that apply register assignment before instruction scheduling. Approaches in which instruction scheduling precedes register assignment are evaluated in Section 5.2. A third strategy, which integrates register assignment and instruction scheduling into a single phase is discussed in Section 5.3. Finally, Section 5.4 evaluates the strategies and gives the direction in which research should go.

5.1 Early Register Assignment

We speak of *early register assignment* or *post-pass scheduling* when register assignment precedes instruction scheduling. This results in an efficient register assignment; i.e., few variables are spilled to memory. However, the register allocator is likely to assign the same register to variables, which are referenced by unrelated operations. The re-use of registers introduces new (false) dependence constraints in the data dependence graph (DDG), making instruction scheduling more restricted.

Historically, the merit of early assignment was that processors offered little exploitable ILP and contained few registers. So, whereas there was much to be lost by poor register assignment, there was little to be gained by good instruction scheduling. Today, however, modern microprocessors contain many registers and provide opportunities to exploit ILP.

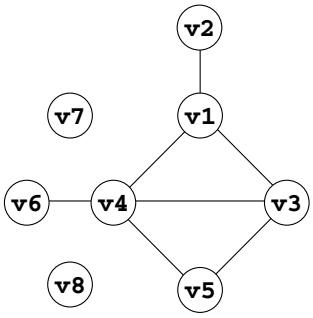
In Section 5.1.1, the limitations of early register assignment in relation to ILP exploitation are discussed. Section 5.1.2 evaluates solutions in literature that try to alleviate this problem. In Section 5.1.3, a practical implementation as used in the TTA compiler back-end is discussed. Section 5.1.4 summarizes the presented methods and gives an evaluation of the implemented method.

5.1.1 ILP and Early Register Assignment

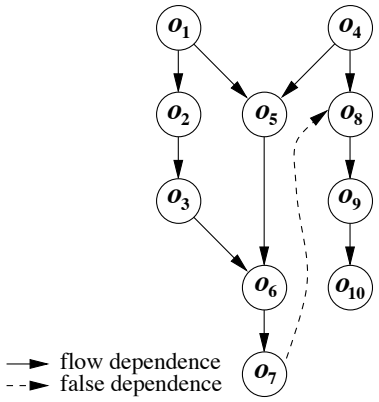
The selection of registers in an early assignment register allocator may limit the possibilities to reorder instructions, due to extra dependences that are introduced with the re-use of registers. These extra dependences are called *false dependences*. A false dependence connects two dependence paths in the DDG;

```
o1    ld   v1
o2    div  v2, v1, #3
o3    add  v3, v2, #10
o4    ld   v4
o5    mul  v5, v1, v4
o6    add  v6, v5, v3
o7    st   v6
o8    mul  v7, v4, #3
o9    add  v8, v7, #2
o10   st   v8
```

a) Code fragment.



b) The interference graph.



c) DDG with a false dependence.

Figure 5.1: Early assignment example.

false dependences can increase the critical path length and the execution time of a program.

Let's look again at the example of Figure 1.4. The corresponding code fragment with operation numbers is shown in Figure 5.1a. The interference graph associated with the code fragment is given in Figure 5.1b. It can be colored with three colors in various ways. Figure 5.1c shows the DDG. Without false dependences the DDG has a critical path of five cycles, under the assumption that each operation takes one cycle. Assuming infinite resources, this code fragment can therefore be scheduled in five instructions. When the variables `v1`, `v5`, `v6`, `v7` and `v8` are mapped onto register `r1`, variable `v2` and `v3` onto `r2` and `v4` onto register `r3`, as shown in Figure 1.4, it is no longer possible to schedule the code fragment in five instructions. This is caused by the false dependence introduced by the register allocator, see Figure 5.1c. The critical

path of the DDG increases to seven cycles, assuming that a read and a write of a register can occur in the same cycle. A mapping of the form `v1` and `v5` onto register `r1`; `v2`, `v3` and `v6` onto `r2`; and `v4`, `v7` and `v8` onto `r3` would not result in a longer schedule, since this register assignment does not increase the critical path of the DDG. From the register allocator's point of view, both register assignments are equally good; however, the latter assignment results in faster executing code.

When not enough registers are available to hold all variables that are live simultaneously, some variables are spilled to memory. This is done before instruction scheduling. The extra inserted instructions can be scheduled in the same way as other instructions. The same idea applies to the code that is responsible for saving the state of a program around procedure calls. The program is analyzed and the necessary code is inserted in the unscheduled (sequential) code. The described method is referred to as *global strictly early assignment* because the registers are assigned to variables for a complete procedure prior to instruction scheduling.

5.1.2 Dependence-Conscious Register Assignment Strategies

Most methods for global early register assignment are based on the work of Chaitin [CAC+81, Cha82]; a number of improvements are published later on [Bri92]. These methods are based on graph coloring and assume that operations do not move relative to one another. However, in the presence of instruction scheduling this assumption is wrong. This observation is the basis of a series of papers that try to extract information from the unscheduled program about operations that may move relative to one another. In the following, approaches are discussed that try to avoid the introduction of false dependences in an early assignment register allocator, thus preserving more ILP enhancing possibilities for the instruction scheduler.

Round-robin Register Selection [HG83, GWC88, BEH91a]
In an attempt to efficiently allocate variables to registers, most register allocators select the first available register for a variable. Registers with a low index are selected first and are re-used more frequently than registers with a high index. As a result, false dependences are primarily associated with low indexed registers. Balancing the variables more equally across all registers, using a round-robin approach, very likely reduces the number of false dependences. This observation is made in several papers [HG83, BEH91a] and is considered to be better than a first-fit approach.

However, a round-robin selection policy of registers does not explicitly take into consideration how false dependences are introduced. As a result, it can add false dependences, which were not present when using a first-fit approach. As observed in [GWC88], no selection policy is uniformly (i.e. for large and small register sets) superior to others in balancing the length of merged dependence paths in the DDG.

Another disadvantage of a round-robin assignment not observed in [HG83, GWC88, BEH91a] is the impact of a round-robin selection policy on the amount of callee-saved code. When more registers are used, more callee-saved code is required. A round-robin selection policy uses, when there are more variables than registers, all registers. A first-fit approach, however, allocates the same set of variables into a much smaller set of registers. In these cases, a first-fit selection policy is preferable.

DAG-Driven Register Allocation [GWC88]
Goodman and Hsu [GWC88] introduced a register assignment method that uses the data dependence graph (denoted as DDG or DAG) of each individual basic block to avoid the introduction of false dependences. Their strategy uses the *width* and *height* of the DDG. The width of a DDG is defined as the maximum number of mutually independent nodes that need a destination register. The height of a DDG is defined as the length of the longest path (i.e. the critical path). The left-edge algorithm is used for assigning registers. When insufficient registers are available, the register allocator will reduce the width of the DDG to be smaller or equal to the number of registers by reusing registers. While the width is reduced, the height may increase since each re-use of registers may merge two independent paths in the DDG into one. This decreases the exploitable ILP and results in a longer schedule.

To minimize the increase in height of the DDG, the register allocator tries to select registers in a manner such that only *redundant* false dependences are introduced. A false dependence is redundant if the ordering between the operations is already enforced by other dependences. When there are no redundant false dependences, the DAG-driven register allocator tries to minimize the growth of the height, by giving priority to the merging of short paths.

The method as proposed by Goodman and Hsu is only able to allocate registers in straight-line code (i.e. a single basic block). DAG-driven allocation does not consider false dependences between operations of different basic blocks, which make this method less suiteable in combination with extended basic block schedulers. Furthermore, constraints, imposed by for example a limited set of FUs, are not considered when reducing the width of the DDG. These constraints would probably already result in a reduction of the width of the DDG and thus the number of required registers. The reported results are based on a limited set of benchmarks. It is shown that the DAG-driven register allocation outperforms local strictly early assignment.

Register Allocation with Instruction Scheduling: a New Approach [Pin93]
In [Pin93] an early assignment method is proposed, which preserves the property that no false dependences are introduced. Therefore, all options for parallelism are kept for the instruction scheduler. The method is based on the Yorktown Allocator [Cha82]. Instead of using an interference graph, a *parallel interference graph* is used for graph coloring. This interference graph is the union of the traditional interference graph $IG = (N_{var}, E_{interf})$ and the false

dependence graph $G_f = (N_{DDG}, E_f)$. The false dependence graph G_f is generated by analyzing the data dependence graph DDG.

Definition 5.1 *The graph $G_t = (N_{DDG}, E_t)$ is a finite undirected graph with N_{DDG} the set of nodes of the DDG. The set E_t is defined as the set of edges of the transitive closure $C(DDG)$ after removal of the direction of the edges.*

Definition 5.2 *For a given basic block the false dependence graph is defined as the undirected graph $G_f = (N_{DDG}, E_f)$. The set E_f is defined as $E_f = \{(n_i, n_j) \mid n_i, n_j \in N_{DDG} \wedge n_i \neq n_j \wedge (n_i, n_j) \notin E_t\}$.*

Observe that the variables $v \in N_{var}$ are defined by nodes $n \in N_{DDG}$. Thus a node $n \in N_{DDG}$ may correspond to a defining operation and to a variable. This relation is used to construct the parallel interference graph.

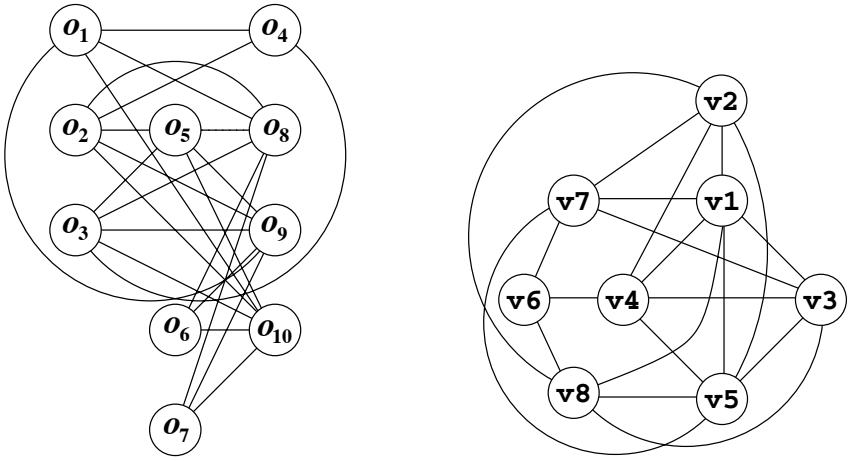
Definition 5.3 *The parallel interference graph, $IG_{par} = (N_{var}, E_{par})$ is a finite undirected graph, with N_{var} the set of variables and E_{par} the set of parallel interference edges: $E_{par} = E_{interf} \cup E_{fdp}$. The set of false dependence prevention edges E_{fdp} is defined as $E_{fdp} = \{(v_i, v_j) \mid (n_{ref(v_i)}, n_{ref(v_j)}) \in E_f \wedge v_i, v_j \in N_{var} \wedge n_{ref(v_i)}, n_{ref(v_j)} \in N_{DDG}\}$ where $n_{ref(v)}$ is a node that references variable v.*

Note that the sets E_{interf} and E_{fdp} may overlap. Due to the sequentiality of the input code, E_{interf} may contain some false dependence prevention edges.

As proven in [Pin93], an optimal coloring of the parallel interference graph provides an optimal register assignment and preserves the property that no false dependences are introduced. However, the number of edges is increased and the probability of finding a legal coloring is reduced. When no valid coloring is found, heuristics are used for trading off parallel scheduling (i.e. the introduction of a false dependence) and spilling. A solution, proposed in [Pin93], is to add weights to the edges of the parallel interference graph that distinguish between those edges that preserve parallelism (E_{fdp}) and those that prevent spills (E_{interf}). The weights should reflect the importance of violating the interference. No examples of how to compute such weights are given in this article.

Figure 5.2a shows the false dependence graph G_f of the code fragment of Figure 5.1. As can be seen the false dependence graph G_f contains many more edges than the data dependence graph DDG of Figure 5.1c. The edges E_f and the interference graph of Figure 5.1b are used to construct the parallel interference graph IG_{par}. This graph is given in Figure 5.2b. At least four registers are required in order to color this interference graph. Adding a false dependence between the operations o_2 and o_5 such that the variables `v1` and `v5` can be mapped onto the same register, reduces the minimal number of required registers to three without increasing the critical path in the DDG.

The method is presented in the context of a basic block, extensions for register assignment and instruction scheduling over multiple basic blocks are provided. However, the constructed parallel interference graph may contain many interference edges that never restrict parallelism. This occurs when references



a) False dependence graph G_f. b) Parallel interference graph IG_{par}.

Figure 5.2: Construction of the parallel interference graph of Figure 5.1.

of involved variables are far apart in the code and thus never will be scheduled in parallel. The large number of interference edges makes coloring difficult. No results are presented to support Pinter's claims that this method improves the quality of the produced code.

Dependence-Conscious Global Register Allocation [AEBK94]
The early register assignment method, proposed by Ambrosch, Ertl, Beer and Krall [AEBK94], is based upon the Optimistic Allocator [Bri92]. In conventional graph coloring [Cha82, Bri92], the interference graph is computed from totally ordered code. This ordering may cause some interference edges that need not to be valid for the final schedule. To avoid this problem, Ambrosch *et al.* compute the interference edges of a basic block b from its DDG. The notions of "before" and "after" in a totally ordered basic block are replaced by the data dependence relations, which is a partial ordering. Into the DDG of b a top node $\top$ is inserted that represents the definition point of all variables that are referenced in b and are elements of the set $live_{In}(b)$. Similarly, a bottom node $\bot$ is inserted that is the use point of all variables referenced in b that are members of $live_{Out}(b)$. The set $N_{Def(v,b)}$ is defined as the set of nodes that define variable v in basic block b and $N_{Use(v,b)}$ is defined as the set of nodes that use variable v in basic block b. The constructed interference graph is denoted as the *minimal* interference graph IG_{min}. This graph only contains interference edges that are present in all possible code orderings.

Definition 5.4 *The minimal interference graph $IG_{min}(b) = (N_{var}, E_{min})$ of a basic block b, is a finite undirected graph, with N_{var} the set of variables and E_{min} the minimal set of interference edges: $E_{min} = \{(v_i, v_j) \mid v_i, v_j \in N_{var} \wedge v_i \neq$*

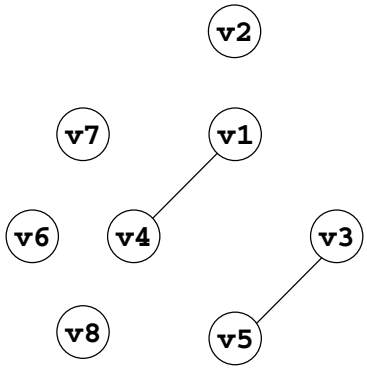


Figure 5.3: The minimal interference graph of Figure 5.1.

$v_j \wedge \exists(n_{def(v_i)}, n_{use(v_j)}) \in C(DDG) \wedge \exists(n_{def(v_j)}, n_{use(v_i)}) \in C(DDG)\}$ *where* $n_{def(v)} \in N_{Def(v,b)}$ *and* $n_{use(v)} \in N_{Use(v,b)}$.

To construct the minimal interference graph for a complete procedure, all graphs $IG_{min}(b)$ of all basic blocks are combined into a single interference graph. Conventional data flow analyses is used for computing the global interferences. In contrast with the conventional interference graph, this graph contains fewer edges because it is not bound to a preordered operation sequence.

The minimal interference graph IG_{min} of Figure 5.1 is shown in Figure 5.3. This graph only contains two interference edges. Both interference edges will be present in any possible code ordering. This graph shows that the code fragment of Figure 5.1 only requires two registers. However, to accomplish this, a false dependence must be added between o_{10} and o_1. This results in a critical path of eight cycles. Note further that in any schedule variable `v2` interferes with `v1` or `v5`.

During coloring, the register selection algorithm is made aware of the false dependences it can introduce. The absence of an edge in the minimal interference graph indicates that coloring a pair of nodes with the same color might introduce a false dependence. This only hurts performance when the introduced false dependence is not redundant. The register selection algorithm avoids introducing non-redundant false dependences, if possible. If this is not possible, false dependences are introduced that connect only short paths in order to minimize the increase of the dependence paths in the DDG. The introduction of a false dependence results in changes of the minimal interference graph. Consequently, each time a false dependence is introduced, the minimal interference graph must be recomputed.

Only preliminary results are published based on two benchmark programs. The results show that the number of interference edges is reduced by 7%–24% and false dependences by 46%–100%. The impact on the execution time of the benchmarks is not listed.

A Scheduler-Sensitive Global Register Allocator [NP93]
Norris and Pollock [NP93] present an approach to build a Scheduler-Sensitive Global register allocator (SSG) based upon Brigg's Optimistic Allocator [Bri92]. The main difference lies in building the interference graph. Norris and Pollock build the interference graph from the data dependence graphs (DDG) of each individual basic block, rather than from the ordering of the intermediate code. The later ordering is usually the coincidental result of some earlier compiler phase. The interference graph $IG_{SSG}(b)$ reflects whether two variables interfere given any legitimate code ordering. As a result $IG_{SSG}(b)$ contains many more interference edges than $IG_{min}(b)$.

Definition 5.5 *The interference graph $IG_{SSG}(b) = (N_{var}, E_{SSG})$ of a basic block b, is a finite undirected graph, with N_{var} the set of variables and E_{SSG} a set of interference: $E_{SSG} = E_A \cup E_B \cup E_C$ where:*

$$
\begin{aligned}
E_A &= \{(v_i, v_j) \mid v_i \in live_{Def}(b), v_j \in live_{In}(b) \wedge v_j \in live_{Out}(b)\} \\
E_B &= \{(v_i, v_j) \mid v_i \in live_{Def}(b), v_j \in live_{In}(b) \wedge v_j \notin live_{Out}(b), \\
&\qquad \exists (n_{use(v_j)}, n_{def(v_i)}) \notin E^T\} \\
E_C &= \{(v_i, v_j) \mid v_i, v_j \in live_{Def}(b), n_k \in N_{def(v_j)},\ \ n_l \in N_{def(v_i)},\ \ k < l \ \wedge \\
&\qquad \exists (n_{use(v_j)}, n_{def(v_i)}) \notin E^T\}
\end{aligned}
$$

where E^T is the set of edges of the transitive closure of the DDG, and $k < l$ indicates that node n_k precedes node of n_l in the preordered operation sequence.

The interference graph IG_{SSG} for the code fragment of Figure 5.1 is in this case equal to IG_{par} and is shown in Figure 5.2b. Both graphs are equal because in this particular situation the set E_A is empty.

Although the interference graph now reflects the maximum freedom of code reordering per basic block, the increased number of interferences will make the register allocator's task more difficult as it will be less able to color the larger interference graph. Norris and Pollock propose to add extra DDG edges in two steps to reduce the number of interferences. First, they add DDG edges prior to building the interference graph. To identify the basic blocks whose DDGs require additional DDG edges, the register requirements per basic block are estimated. In this step, only *scheduler-sensitive* edges are added to the DDG. An edge is scheduler-sensitive if the schedule generated by the instruction scheduler would contain the edge anyway. This can be the result of other dependences or resource constraints. Adding only scheduler-sensitive edges may not be sufficient to create a colorable graph. To handle these situations, in a second step additional DDG edges are added during register assignment. When no legal coloring can be found, a node of the interference graph is selected with the greatest number of interferences that can be eliminated by adding false dependence edges to the DDG. If there are not enough possibilities to eliminate interferences so that the node will be colorable, no DDG edges

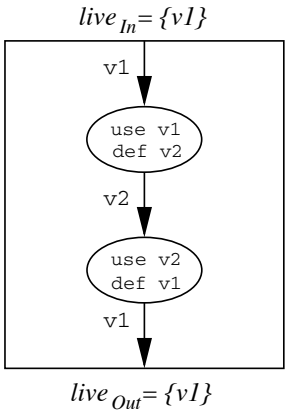


Figure 5.4: Non-interference that results in an interference edge in E_{SSG}.

are added and the node with the least spill costs is selected for spilling. Their experiments show a significant improvement over global strictly early register assignment for Livermore loops.

Examination of the presented algorithm shows that the construction of IG_{SSG} is too conservative. The set E_{SSG} contains more edges (and thus interferences) than necessary. This is illustrated in Figure 5.4. Variable `v1` is live on entry and exit of the basic block, but will never interfere with variable `v2`. The set E_{SSG} contains, however, the interference edge $(v1, v2)$ because this edge is a member of the set E_A.

Another disadvantage of this method is that only the false dependences within basic blocks are considered. As a result, a region scheduler will be constrained by inter basic block false dependences. To extend this method for region scheduling, one should compute the IG_{SSG} for a region instead of a basic block. This, however, results in many more interference edges, which makes coloring hard.

5.1.3 Dependence-Conscious Early Register Assignment for TTAs

The previous paragraphs showed that avoiding false dependences, or placing them where they cannot hurt performance, is an important technique to enhance performance. The TTA instruction scheduler operates on regions. Consequently, methods that have the ability to avoid potential false dependences between operations in the same basic block, and between operations in different basic blocks, are required to exploit a large amount of ILP. The early register allocator used in this thesis is based on the work of Pinter [Pin93]. Instead of

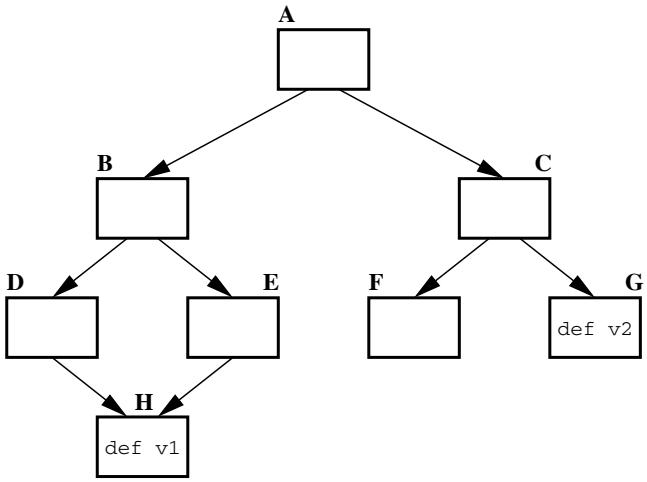


Figure 5.5: CFG of code fragment.

the Yorktown Allocator [Cha82], the Optimistic Allocator [Bri92] is used, because of it's better performance.

Pinter's parallel interference graph may represent interferences that almost never restrict parallelism. This occurs when references of involved variables are far apart in the code and thus never will be scheduled in parallel. This is illustrated in Figure 5.5, which shows the CFG of a code fragment. A false dependence between the definition of variable `v1` and the definition of variable `v2` only restricts ILP when both operations are imported in basic block **A**. However, this is very unlikely because of resource constraints and dependences on other operations. Pinter's parallel interference graph reflects this dependence. This makes coloring hard and may even result in spilling, or the introduction of more restrictive false dependences.

Based on this observation, the method to build the false dependence graph is slightly modified [Hoo96]. The new constructed graph is called the *forward false dependence graph* $G_{ff} = (N_{DDG}, E_{ff})$. To avoid too many false dependence edges, only potential *forward* false dependences are recorded. A forward false dependence is a false dependence between operations between which a control flow path exists. The G_{ff} is defined as:

Definition 5.6 *The forward false dependence graph,* $G_{ff}(P) = (N_{DDG}, E_{ff})$ *of a procedure* P *is a finite undirected graph, with* N_{DDG} *the set of operations and* E_{ff} *the set of forward false dependence edges:* $E_{ff} = E_f - \{(n_i, n_j) \mid bb(n_i)\ \neg doms\ bb(n_j) \wedge bb(n_j)\ \neg doms\ bb(n_i) \wedge n_i, n_j \in N_{DDG}\}$ *where* $bb(n)$ *is the basic block of operation* n *and* $doms$ *gives the dominance relation between basic blocks.*

The forward parallel interference graph is constructed by combining the edges of E_{ff} and E_{interf} in the same way as Pinter suggests.

Definition 5.7 *The forward parallel interference graph,* $IG_{fpar}(P) = (N_{var}, E_{fpar})$ *of a procedure* P *is a finite undirected graph, with* N_{var} *the set of variables and* E_{fpar} *the set of forward parallel interference edges:* $E_{fpar} = E_{interf} \cup E_{ffdp}$*. The set of forward false dependence prevention edges* E_{ffdp} *is defined as* $E_{ffdp} = \{(v_i, v_j) \mid (n_{ref(v_i)}, n_{ref(v_j)}) \in E_{ff} \wedge v_i, v_j \in N_{var} \wedge n_{ref(v_i)}, n_{ref(v_j)} \in N_{DDG}\}$.

The forward parallel interference graph IG_{fpar} for the code fragment of Figure 5.1 is in this case equal to IG_{par} because the false dependences do not cross basic block boundaries. However, the forward parallel interference graph for a complete procedure reflects fewer potential false dependences than the original parallel interference graph as proposed by Pinter. Consequently, the forward parallel interference graph is easier to color. Also the following observation justifies the choice only to consider forward false dependences. To achieve high performance, the operation selecting heuristics of the instruction scheduler will favor operations from control flow paths with a high execution probability. Because most branches are biased towards one direction [PSM97], operations will be selected from the same control flow path. Therefore, false dependences between operations in the same control flow path will hurt the attainable performance more than false dependences between operations in different control flow paths.

When insufficient registers are available, the register allocator has to decide whether to spill a variable or to add a false dependence. The method proposed in [Hoo96] always chooses for the latter if possible. In order to decide which false dependence to introduce, each interference edge is augmented with a weight. This weight reflects the possible negative effect on performance when two variables v_i and v_j are assigned to the same register.

$$W_{ffdp}(v_i, v_j) = \max\left(\frac{f(m_{ref(v_i)}) \cdot Pr(m_{def(v_j)} | m_{ref(v_i)})}{dist(m_{ref(v_i)}, m_{def(v_j)})}\right) \tag{5.1}$$

The move $m_{ref(v_i)}$ uses or defines variable v_i and move $m_{def(v_j)}$ defines variable v_j. The weight is proportional to the execution frequency f of $m_{ref(v_i)}$ and the probability that $m_{def(v_j)}$ will be executed after $m_{ref(v_i)}$. The weight is inverse proportional to the number of moves (the distance) between $m_{ref(v_i)}$ and $m_{def(v_j)}$ in the sequential code. Because there can be multiple combinations of potential false dependences between two variables, the maximum weight of all combinations is taken. A high weight indicates that avoiding the associated false dependence is important and will probably result in higher performance of the generated schedule. When the register allocator cannot find a proper node coloring, it introduces a false dependence with the lowest weight.

In the remainder of this thesis, the method described in this section is referred to as *DCEA* or *Dependence-Conscious Early Assignment*.

5.1.4 Discussion, Experiments and Evaluation

In the previous sections, various dependence-conscious register assignment strategies were described. The round-robin approach distributes the registers in a round-robin fashion to the variables hoping that false dependences are not introduced. This does not seem to be a constructive method. All other discussed methods use the DDG to identify whether a particular assignment will result in a false dependence. Goodman and Hsu [GWC88] assign registers with the use of a left-edge algorithm, without introducing false dependences. When insufficient registers are available, false dependences are added in such a way that the impact on the total schedule is minimized. This method can only be applied to straight-line code. The method as proposed by Norris and Pollock [NP93] is too conservative, the method as proposed by Ambrosch et al. [AEBK94] is computational intensive and the method of Pinter [Pin93] results in graphs that are hard to color (the same holds for the method as proposed by Norris and Pollock). The method as used by Hoogerbrugge [Hoo96] does not have these problems, however, it may ignore important false dependences.

The approaches of Norris and Pollock, Ambrosch et al., Pinter and Hoogerbrugge are closely related. All are based on graph coloring. The following relations are identified assuming all methods operate on the same scheduling scope.

$$E_{min} \subseteq E_{interf} \subseteq E_{fpar} \subseteq E_{par} \tag{5.2}$$

$$E_{ffdp} \subseteq E_{fdp} \tag{5.3}$$

The graph $IG_{SSG}(b)$ does not consider false dependences between operations in different basic blocks. When we restrict the scheduling scope to a basic block, the following relation is identified.

$$E_{min} \subseteq E_{interf} \subseteq E_{fpar} = E_{par} \subseteq E_{SSG} \tag{5.4}$$

It should be noted that E_{SSG} is the largest set of edges. It even contains edges that are not interference edges. Observe that the sets E_{min} and E_{interf} are not equal. The set E_{min} only contains interferences that are present in all possible code orderings, while E_{interf} also can contain additional edges from the set E_{fdp}. Note further, edges present in E_{min} also can be present in E_{fdp}. For example, the edge $(\texttt{v1}, \texttt{v4})$ in the parallel interference graph of Figure 5.2b originates from the set E_{fdp} (and E_{interf}) and is also present in the minimal interference graph of Figure 5.3.

Experiments are performed to evaluate the importance of dependence-conscious early register assignment. Strictly early assignment is compared with DCEA using the region scheduler. Global register assignment is used, e.g. registers are assigned for a complete procedure. For all applications in the benchmark suite (see Section 4.1) the performance gain is measured. The results of these measurements for both target TTAs are shown in Figure 5.6. The

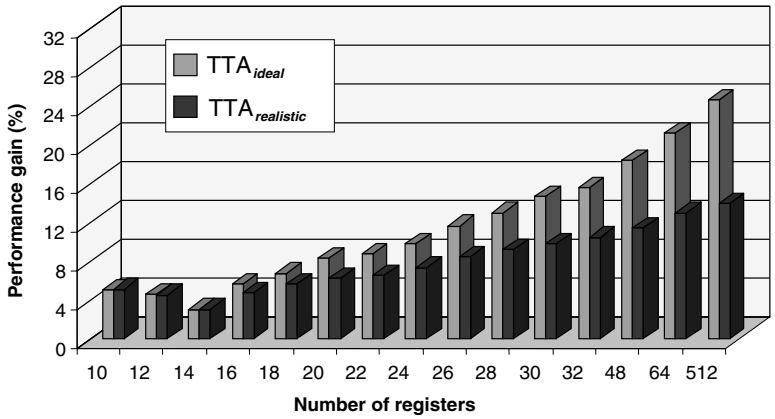


Figure 5.6: Speedup of DCEA compared with strictly early assignment.

number of integer registers is varied and is listed along the x-axis. The speedup of DCEA is listed along the y-axis. The results clearly show that DCEA outperforms strictly early assignment. The performance gain decreases when the number of registers decreases. When a large number of registers is available all potential false dependences can be avoided and thus the impact of DCEA is high. When registers are scarce, it is no longer possible to avoid all potential false dependences, and DCEA has to decide, which false dependences to avoid or to introduce. Consequently, the false dependences introduced by DCEA resulted in a smaller performance gain.

It is also interesting to note that the performance gain of the TTA_{ideal} processor is larger than the performance gain of the $\text{TTA}_{realistic}$ processor. Because strictly early assignment introduces many false dependences, the instruction scheduler is not capable to exploit the large number of resources provided by the TTA_{ideal} processor. As a result, the performance of the TTA_{ideal} processor is only slightly higher than the performance of the $\text{TTA}_{realistic}$ processor when using strictly early assignment. On the other hand, DCEA leaves many code reordering possibilities to the instruction scheduler. Especially, when a large number of registers is available the instruction scheduler is able to use as many resources of the TTA_{ideal} processor as needed. Consequently, the performance difference between the TTA_{ideal} and the $\text{TTA}_{realistic}$ processor is substantial when using DCEA. Both effects explain the larger performance gain of the TTA_{ideal} processor when we compare DCEA with strictly early assignment.

Based on the above discussion one would expect that using fewer registers would result in a smaller performance gain. However, Figure 5.6 shows a minimum around 14 registers. This strange effect can be explained by the observation that for some benchmarks, when using strictly early assignment, the performance decreases significantly when the number of registers drops below 14 registers. This is caused by the introduction of a significant amount

of spill code. The introduced short live ranges are causing extra false dependences. Both early assignment methods are faced with this problem. However, DCEA is still capable to avoid false dependences and to keep the performance degradation limited.

The performance gain of a dependence-conscious early register assignment method is caused by its ability to identified potential false dependences. However, it is difficult to predict which false dependences are really important. It may happen that a false dependence between two operations is avoided that did not limit the available ILP. In these situations, avoiding a different false dependence would be more profitable.

An additional problem with early assignment approaches within the context of TTAs, is their inability to take advantage of software bypassing and dead-result move elimination. These techniques can eliminate the need of some register file accesses; see for example the following code fragment:

```
r1 → add.o; r2 → add.t;
add.r → r3;
r3 → sub.o; r4 → sub.t;
sub.r → r5
```

The scheduled version may turn out not to use register `r3`, because the result of the addition is bypassed to the subtraction and never used again.

```
r1 → add.o; r2 → add.t;
add.r → sub.o; r4 → sub.t;
sub.r → r5
```

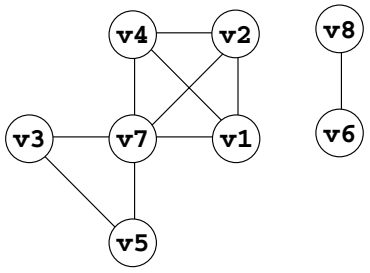
From [Hoo96] it is known that more than 35% of the register file accesses are eliminated. The registers assigned to these variables could be used, when this was known in advance, to avoid spilling or to avoid the introduction of false dependences. An early assignment register allocator has, however, no idea which register references will be eliminated by the scheduler. Whereas software bypassing decreases the register pressure in the scheduled code, the early assignment method cannot exploit this advantage.

5.2 Late Register Assignment

When register assignment is performed after instruction scheduling, i.e. *late register assignment* or *pre-pass scheduling*, the scheduler, uninhibited by false dependences, can generate an efficient schedule. However, the instruction scheduler, in its attempt to reorder instructions to maximize ILP, may lengthen the live ranges of values and thus increases the contention for registers. If not enough registers are provided by the target processor, the data is written to memory, introducing spill code which itself also requires registers. The increase in ILP can be nullified by the amount of spill code.

o_1 :ld v1	o_4 :ld v4
o_2 :div v2, v1, #3	o_8 :mul v7, v4, #3
o_3 :add v3, v2, #10	o_5 :mul v5, v1, v4
o_6 :add v6, v5, v3	o_9 :add v8, v7, #2
o_7 :st v6	o_{10} :st v8

a) Schedule for a 2-issue processor.



b) Associated interference graph.

Figure 5.7: Late assignment example.

In Section 5.2.1, the limitations of late register assignment in relation to ILP are described. Section 5.2.2 discusses various approaches proposed in literature to solve the problems related to ILP and late assignment. Section 5.2.3 presents the late register allocator as implemented in the TTA compiler backend and Section 5.2.4 gives an evaluation of late assignment in the context of TTAs.

5.2.1 ILP and Late Register Assignment

Performing instruction scheduling prior to register assignment has the advantage that the available ILP can be exploited without constraints imposed by register assignment. However, applying late assignment may result in a large register pressure since multiple variables are becoming live simultaneously.

Let's return to the example of Figure 1.4b. Figure 5.7a shows the scheduled version of the code fragment for a 2-issue processor, when instruction scheduling is uninhibited by register assignment. The code fragment executes in five cycles. The associated interference graph is shown in Figure 5.7b. As can be easily seen, at least four registers are required to color the graph. The scheduler has reordered the code in such a way that the register pressure is increased compared to the sequential code of Figure 1.4b. Note that switching the operations o_5 and o_8 results in a register pressure of three. From the instruction scheduler's point of view, both schedules are equally good, however, the latter results in a better overall schedule when only three registers are available.

When only three registers were available, the schedule shown in Figure 5.7a

ld r1	ld $r3_a$
st $r3_a$	
div r2, r1, #3	mul $r3_b$, $r3_a$, #3
st $r3_b$	
ld $r3_a$	
add r2, r2, #10	mul r1, r1, $r3_a$
ld $r3_b$	
add r1, r1, r2	add r2, $r3_b$, #2
st r1	st r2

a) Schedule with inserted spill code.

ld r1	ld $r3_a$
	st $r3_a$
div r2, r1, #3	mul $r3_b$, $r3_a$, #3
st $r3_b$	ld $r3_a$
add r2, r2, #10	mul r1, r1, $r3_a$
	ld $r3_b$
add r1, r1, r2	add r2, $r3_b$, #2
st r1	st r2

b) Rescheduled code.

Figure 5.8: Late assignment and spilling.

requires spill code. The spill code generated by the register allocator is inserted in already scheduled code as shown in Figure 5.8a. Inserting new instructions into the compacted code could violate the constraints under which the code was originally scheduled (this certainly holds for TTAs as we will see later). Rescheduling is usually applied to efficiently integrate the spill code within the schedule, see Figure 5.8b. However, rescheduling may rearrange the code completely. The false dependences introduced by the late register allocator may restrict the new code reordering. This may lead to less efficient schedules. Instead of adding spill code into the already scheduled code, Sweany and Beaty [SB90] proposed to add the spill code to the original unscheduled code without assigning registers to variables. The resulting code is scheduled again and may result in efficient code since the scheduler is never hindered by false dependences. Rescheduling, however, does not guarantee that the newly scheduled code is colorable. Consequently, additional spill code is inserted. This process is repeated until a legal register assignment is found.

Sweany and Beaty [SB90] also observed that insertion of state preserving code is difficult in late assignment approaches. This causes a phase ordering problem, because, until after register assignment, it is not known how many registers need to be saved and restored. The method as proposed for inserting spill code does not apply, because scheduling of state preserving code can lead to other register usage patterns. This may result in the need for more, fewer or

different state preserving code. The solution they propose is to consider all registers callee-saved. After register assignment, new entry and exit basic blocks are added to each procedure. The necessary callee-saved code is inserted in the new created basic blocks and is scheduled using a local scheduler.

5.2.2 Register-Sensitive Instruction Scheduling Strategies

The simplest late assignment approach is to apply instruction scheduling and register assignment in two separate phases without any form of communication between the two phases, called *strictly late assignment*. This implementation is quite straightforward, but the instruction scheduler may produce schedules that require more registers than available. The approaches discussed in this section add extra heuristics to the instruction scheduler in order to reduce the register pressure.

Integrated Prepass Scheduling [GWC88]
Goodman and Hsu presented a method, called *integrated prepass scheduling (IPS)*, which performs late register assignment, but attempts to restrict the number of concurrently live local variables by giving each basic block a register limit. The register limit places an upper bound on the number of live local variables, thus limiting the amount of ILP that the local instruction scheduler may exploit. The instruction-based list scheduler selects operations to exploit ILP, unless the number of live local variables is greater or equal to the given limit. The scheduler then tries to schedule operations that reduce the number of simultaneously live local variables. By keeping track of the number of available registers, the scheduler can choose the appropriate scheduling technique to produce a better code sequence. The initial number of available registers per basic block is determined by the total number of registers, minus the number of global registers live-on-entry of the basic block. The method combines two scheduling techniques, one to exploit ILP and the other to minimize register usage, into a single phase. After scheduling, a local register allocator assigns registers to the variables within the basic block; spills to memory are inserted if the limit could not be met. The proposed method assumes that global variables are already assigned to registers. Furthermore, no attempt is made to schedule inserted spill code efficiently.

The reported results are based on a limited set of benchmarks (the first twelve Livermore loops) and showed improvements in the order of 15% for 10 registers compared to strictly late assignment. The method heavily depends on the availability in the *ready* set of operations that can decrease the number of simultaneous live registers. Goodman also observed that late assignment often resulted in register spilling. Therefore, the scheduled programs had significant larger sizes than programs produced with an early assignment approach. This makes late assignment less suitable for application specific processors, since often the code size is a critical design parameter.

A variation of Integrated Prepass Scheduling [BEH91a]
The method proposed by Bradlee, Eggers and Henry is a variant of Goodman and Hsu's Integrated Prepass Scheduling (IPS) [GWC88]. The main improvement is the use of a global register allocator. To model reserved registers for important global variables, Bradlee's IPS sets the local register limit for each basic block to the maximum number of available registers. This limit is reduced by the number of unique global variables referenced within the basic block, instead of assigning global variables prior to running IPS as done in [GWC88]. Instruction scheduling is applied per basic block. It tries to generate code within the local register limit. After scheduling, the variables in the scheduled code are assigned to registers by the Chaitin's global register allocator [Cha82]. A local post-pass scheduler is invoked after register assignment to ensure that spill code is scheduled as well as possible. The reported results showed that for 32 registers this variation of IPS produces code that is on average 13% faster than strictly early assignment.

Like the method of Goodman, this method also applies instruction scheduling per basic block. The reported results show that the proposed method produces code that is on average faster than a strictly early assignment method. However, the results are based on comparison with a non-dependence-conscious early register allocator.

The (α, β)-Combined Heuristic [MPSR95]
Motwanu et al. propose a heuristic, which combines controlling register pressure and instruction-level parallelism considerations. Prior to scheduling, an ordering of the operations is determined. The priority function, which determines the ordering, consists of two parts: (1) the schedule rank γ_S for which a priority function of any good list scheduling can be chosen, and (2) the register rank γ_R defined as:

$$\gamma_R(o_i) = \min_{o_j \in succ_v(o_i)} \max \left\{ L_{path}(o_i, o_j), \frac{T_{path}(o_i, o_j)}{\mid FU_{set} \mid} \right\} \quad \forall\, o_i \in N_{DDG} \qquad (5.5)$$

where $L_{path}(o_i, o_j)$ represents the distance in the DDG between the operations o_i and o_j, $T_{path}(o_i, o_j)$ is defined as the total path length in the DDG of all paths from o_i to o_j, $succ_v(o_i)$ the set of all successors of o_j in the DDG that read variable v which is referenced by o_i, and $\mid FU_{set} \mid$ represents the number of function units. The register rank is zero when $succ_v(o_i) = \emptyset$. This priority function favors operations that use variables in short live ranges. Scheduling these uses close to their definition reduces the register pressure.

The combined rank function $\gamma = \alpha \cdot \gamma_S + \beta \cdot \gamma_R$ orders the operations into a list in increasing order of rank. With the parameters α and β the algorithm is tuned. These parameters obey the following equality $\alpha + \beta = 1$. Without worrying about the register bound, a greedy local list scheduling algorithm uses the ordered list to obtain a schedule. Afterwards the code is checked for over-using registers and spill code is inserted.

```
r1 → add1.o; r2 → add1.t;
r4 → add2.o; r5 → add2.t;
add1.r → r3;
add2.r → r6;
```

a) Original schedule.

```
r1 → add1.o; r2 → add1.t;
r4 → add2.o; r5 → add2.t;
...
add1.r → r3;
add2.r → r6;
```

b) Schedule with inserted instruction.

Figure 5.9: Instruction insertion problem.

Instead of verifying the algorithm with real programs, Motwanu verifies the algorithm with randomly generated DDGs of unrealistic large basic blocks (100 operations). The experiments showed that the (α, β)-combined heuristic outperforms on average strictly late assignment with 16% and strictly early assignment with 4% when 16 registers were available.

In contrast to IPS, this method does not switch abruptly from selection priority, but uses a smoother transition to decide whether reducing register pressure is more important than increasing ILP. However, the proposed method accomplishes this by assigning prior to scheduling a static priority to the operations; it does not adapt its operation selection criteria during scheduling. Since the results were obtained by using synthetic benchmarks, they cannot be straightforwardly extrapolated to real programs.

5.2.3 Register-Sensitive Instruction Scheduling for TTAs

In the context of TTAs, early assignment has the drawback that it is unable to exploit the registers saved by software bypassing and dead-result move elimination. Late assignment, however, can exploit this TTA specific optimization. Since fewer variables exist in the scheduled code, it is more likely to find a legal register assignment. This results in an interesting observation. Late assignment for TTAs results in more spill code because of the increase of the live spans, but on the other hand it reduces the number of spills due to the ability to exploit the benefits of software bypassing and dead-result move elimination.

Unfortunately, late assignment has additional problems in the context of TTAs. The first problem is called the *instruction insertion problem* [Cor98]. Because operation latencies are visible, the insertion of instructions in already scheduled code may violate the constraints under which the code was originally scheduled. This is shown in Figure 5.9. Assume both additions are executed on the same VTL pipelined FU, which has a latency of two cycles. Inserting an instruction as is done in Figure 5.9b has as a consequence that both operations produce the same result. This occurs because the result register in the FU is overwritten by the second operation before the first operation could read it. Note, that this problem does not arise when hybrid pipelined FUs were used.

Independent of the type of FU pipelining there is another problem related

```
r1 → ld.t;
v1 → add.o;
ld.r → r3;
```

a) Schedule prior to spilling.

```
r1 → ld.t;
spill_address → ld.t;
ld.r → add.o;
ld.r → r3;
```

b) Schedule after spilling.

Figure 5.10: Spill code insertion for TTAs.

to inserting spill code; this problem is illustrated in Figure 5.10a. Suppose variable `v1` must be spilled. In this case, a load operation is required to read the value of `v1` from memory. However, simply inserting a load operation will result in an invalid schedule, see Figure 5.10b. The operation is inserted in the middle of another operation. This disturbs the pipeline of the FU; the original load will receive a value from an incorrect memory location. Note, that this problem can be alleviated when both load operations are executed on different FUs. This requires, however, a TTA that has at least one free FU that can perform a load operation.

The last problem mentioned in relation to the insertion of spill code is rescheduling. In contrast to conventional processors, where after register assignment and spilling all variables reside in registers or memory, in TTAs variables are also hidden by software bypassing and dead-result move elimination. The hidden variables might reappear in the rescheduled code, however, they are not mapped onto a register. Rescheduling might also result in an opposite effect; variables that were mapped onto registers become hidden because of software bypassing and dead-result move elimination. A solution to solve this problem is to use the approach described in [SB90]. After register assignment, spill code is inserted in the original unscheduled code. To the unscheduled code, no software bypassing and dead-result move elimination optimizations are applied. Consequently, the code with the inserted spill code can be scheduled in the same way as the original code. This process is repeated until no spill code is required. Note that schedulers with large scheduling scopes (extended basic block schedulers) rearrange code drastically and due to the insertion of spill code the schedule will not resemble the first version. Spill code inserted in the early iterations might be void in the eventual schedule. Another drawback is the long compile time to generate the schedule. Despite of the disadvantages mentioned above, the proposed method is implemented in the TTA compiler back-end because it seems to be the only valid approach to generate code in the context of late assignment.

Inserting state preserving code gives another problem. As was observed in [SB90] the insertion of caller- and callee-saved code is in itself a phase ordering problem. The solution proposed by Sweany is to save all used registers on entry and on exit of a procedure by adding extra basic blocks. Thus, all registers are callee-saved. This approach is also applicable to the TTA late assignment approach. Note, however, that this may lead to inefficient schedules

since it is no longer possible to make a trade-off between caller- and callee-saved registers. Furthermore, the extra inserted basic blocks are scheduled by a local scheduler. This leads to a reduction of the exploitable ILP.

To prevent the insertion of too much spill code, we tried to limit the greediness of the scheduler. In the previous section, register sensitive scheduling methods are described that keep track of the number of available registers. The scheduler switches between a selection heuristic that exploits ILP, and a heuristic that favors operations that reduce the register pressure. Key to these approaches is an accurate estimate about the registers required when selecting an operation for scheduling. This can easily be done in an instruction-based list scheduler; however, for an operation-based list scheduler, as used in our compiler back-end, this is problematic. Each instruction in the partial schedule has its own different register limit. Because it is not known in which instruction an operation will be scheduled, it is also not known which register limit to use. In our experiments, we favored operations that free registers when it turned out that scheduling the most critical operation in the $ready$ set increases the register pressure above a certain threshold. Unfortunately, on average no performance improvements were found. This is caused by various reasons: (1) it is not possible to make a good register pressure estimate, (2) the $ready$ set is small, which reduces the probability of finding a register pressure reducing operation, (3) changing the priority function, which favors operations in the critical path, has a negative impact on performance and (4) *floaters*. Floaters are operations that do not depend on other operations in a basic block and when unhindered by false dependences *float* to the top of a basic block. When these floaters define variables, they increase the register pressure.

5.2.4 Experiments and Evaluation

In the previous sections, various register-sensitive instruction scheduling strategies were described. These strategies try to limit the register pressure when scheduling. The methods known from literature are all based on instruction-based list scheduling. Scheduling for TTAs, however, requires operation-based list scheduling. Our experiments showed that applying the methods for instruction-based list scheduling in a TTA instruction scheduler did not result in any improvement.

Most of the methods known from literature can only operate on single basic blocks. Our experiments indicate that DCEA outperforms late assignment on average with 5% when using a local scheduler. However, in order to exploit larger amounts of ILP, larger scheduling scopes should be considered as well. The performance gain of DCEA over late assignment, when using region scheduling, is shown in Figure 5.11. The performance penalty of late assignment is large. The scheduler in a late assignment strategy is not constrained by false dependences. It imports as many operations as permitted by other resources constraints. This results in a huge amount of variables that are simul-

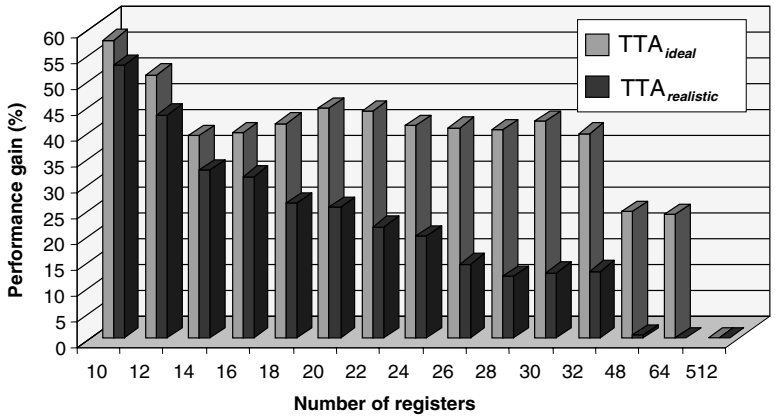


Figure 5.11: Speedup of DCEA compared with late assignment in the context of region scheduling.

taneously alive. When not enough registers are available this results in a large amount of spill code. The performance difference between the two benchmark TTA processors can be explained by the following. For the $\text{TTA}_{realistic}$ processor, due to resource constraints other than registers, the scheduler cannot apply importing as aggressively as for the TTA_{ideal} processor. Consequently, the schedules generated for a TTA_{ideal} processor contain more simultaneous live variables. This results in a higher register pressure and thus more spill code is required. This reduces the performance. Despite our effort to find heuristics to improve the performance of late assignment, early assignment still generates faster executing code. This behavior of late assignment was also observed in [BSBC95] where it was stated that there are times when spilling dramatically increases the execution time well beyond any scheduling gain obtained by late register assignment.

5.3 Integrated Register Assignment

As we have seen in the previous sections, the division of instruction scheduling and register assignment into separate phases can affect the performance of these tasks and thus the quality of the generated code. Both discussed approaches, early and late assignment, have problems. In effect, the lack of communication and cooperation between the instruction scheduler and the register allocator can result in code that contains excess register spills and/or lower degree of instruction-level parallelism than possible. Improved performance in one phase can deteriorate the performance of the other phase, possibly resulting in poorer overall performance.

In this section, various approaches from literature are discussed that address integrated register assignment and instruction scheduling. These ap-

proaches can be divided into three classes: the first class invokes register assignment and instruction scheduling multiple times for a complete procedure. These methods are discussed in Section 5.3.1. The second class uses integer linear programming to address the phase ordering problem [EGS95, CCK97]. We chose not to discuss them because these methods tend to have long compilation times. Their practical use is limited to straight-line code or small loops only. The third class truly integrates both phases into a single phase; these approaches are discussed in Section 5.3.2.

5.3.1 Interleaved Register Assignment

Interleaved register assignment and instruction scheduling strategies apply both phases multiple times to get correct estimates of the expected constraints imposed by one phase to the other phase. Only a few approaches are known which apply this strategy. This is probably caused by the long compilation times required for these methods.

Register Allocation with Schedule Estimates [BEH91a]
The strategy proposed by Bradlee *et al.* [BEH91a], called *RASE* (*Register Allocation with Schedule Estimates*), consists of three steps. The first step performs multiple times local early register assignment, followed by local instruction scheduling with a varying number of registers. A *schedule cost estimate* is computed for each number of registers. This schedule cost estimate is the estimated number of cycles required to execute the basic block while remaining within a certain register limit. In the second step, a global register allocator (based on the work of Chaitin [Cha82]) partitions the register set for each basic block into two sets. One set is used for global variables and the other set is the basic block's register limit. Based on the spill costs and the schedule cost estimate, the global register allocator determines the appropriate balance between these two competing needs for registers. The third step schedules each basic block and assigns registers to variables within the basic block's register limit in the same way as suggested in [GWC88]. Spill code is inserted when insufficient registers are available for the local live ranges. The reported results showed that RASE produced code is on average 8% faster than strictly early assignment for Intel's i860. A drawback of RASE is that it can only be applied to basic block scheduling, because otherwise the register limit per basic block loses its meaning.

Combining Register Assignment Interference Graphs [BSBC95]
Brasier *et al.* [BSBC95] describe in their paper a framework, called *CRAIG* (*Combining Register Assignment Interference Graphs*) that combines register assignment and instruction scheduling to tackle the phase ordering problem. CRAIG performs first late assignment. The generated schedule is not hindered by register assignment and exploits all available ILP. The register allocator is invoked to compute the *late interference graph*. The generated schedule is accepted when

no spilling is required. Otherwise, CRAIG constructs an *early interference graph* for the original unscheduled code. This graph generally has fewer interference edges than the late interference graph. When spill code is needed to color the early interference graph, CRAIG inserts spill code and invokes the scheduler, based upon the assumption that this is the best that can be done under the circumstances. Otherwise, it assumes that it is likely that false dependences have been added by the register allocator, and thus, the resulting schedule can be improved. CRAIG will attempt to reclaim some of this lost efficiency by removing as many of the false dependences as possible, up to the point where spilling is needed. By adding edges to the early interference graph that are found exclusively in the late interference graph, CRAIG creates interference between those values, which the scheduler forced to be in different registers. If they were mapped onto the same register in the early interference graph, then a false dependence that potentially inhibits a more efficient schedule is identified and removed.

The method is applied to a limited set of benchmarks. When registers are scarce the results showed an average performance increase of 6.7% compare to strictly early assignment, and 3.9% compared to strictly late assignment. The described method uses a random approach towards selecting edges that can remove false dependences. It is observed that more accurate selection criteria must be found to increase the performance. The proposed algorithm stops removing false dependences when spill code is required. As a result, it can occur that not all false dependences that prevent the generation of an efficient late assignment schedule are removed. This may result in a schedule in which originally not recorded false dependences restrict parallelism due to a different scheduling order of the operations than the scheduling order that was used to compute the late interference graph.

5.3.2 Integrated Instruction Scheduling and Register Assignment

There are obvious pros and cons to doing register assignment early or late. Because register assignment and instruction scheduling are antagonistic, it seems profitable to merge both phases into a single step. In the past, the integration of the instruction scheduling and the register assignment has been considered as too complicated [BEH91a]. However, due to the ongoing research to exploit more and more ILP, the register pressure will increase and registers should be assigned in such a way that exploitation of ILP is not hindered.

A Unified Resource Allocator [BGS93]
The *URSA* (*Unified ReSource Allocator*) presented by Berson *et al.* [BGS93] unifies the problems of allocating registers and function units. This technique operates on the DDG of the program. The purpose of the URSA is to modify the DDG in such a way that its resource requirements cannot exceed the capacity of the target machine. Therefore, it is only concerned with the allocation of resources,

and not their actual assignment. The first phase carries out the measurement of resource requirements and identifies regions with excess requirements. A so-called *re-use* directed acyclic graph (DAG) is constructed for every resource. These DAGs are used to determine the maximum number of resources of a specific type to obtain an optimal schedule. The second phase applies transformations to the DDG that reduce the requirements to levels supported by the target machine. These transformations add sequential dependence edges to the DDG that remove excess resource requirements. The transformation that is best with respect to the combination of minimizing the critical path and reduction of excess requirements is selected and applied. These extra edges introduce sequentiality, i.e. reduce the exploitable ILP. When it is not possible to reduce the register requirements by adding sequential dependence edges, spill code is introduced. Resource assignment and instruction scheduling follow the DDG transformations. The assignment phase is also responsible for handling any excessive requirements that were not identified by URSA's heuristics. URSA requires a large number of representations to expose the availability of resources. Furthermore, no experimental results are presented to give an indication of the method's effectiveness.

The URSA is defined for local scheduling. In [BGS94] resource spackling, an extension of the URSA, which also supports global code motion, is presented. Resource requirement measurements are used for finding areas where resources are either under or over utilized, called *resource holes* and *excessive sets*, respectively. Conditions for code motion are established to increase the resource utilization in the resource holes and to decrease the resource requirements in excessive sets. These conditions are applicable to both local and global code motion. The results are, however, disappointing, the improvements of global over local instruction scheduling are on average 5.5% while other approaches have shown much larger improvements (e.g. 135%, see Section 4.3).

Integrated Register Assignment in the Bulldog Compiler [Ell86]
The approach described by Ellis [Ell86] integrates register assignment and trace scheduling. Registers are assigned to variables by an instruction-based list scheduler as it produces code for a trace. Since trace scheduling starts scheduling on the crucial traces first, the trace scheduler, which uses a pool of registers, takes as many registers from the pool as it requires. When the trace is scheduled, the register locations of the variables are recorded at every entry and exit of the trace. Later traces adjoining the exits and entries are advised to use these locations. Traces are scheduled as independent entities, therefore it is not always possible to keep a variable in all traces in the same register. To guarantee correct execution, repair code is required.

Ellis showed that trace scheduling makes it hard to manage registers effectively; a register written to an operand of an operation must be considered occupied until the operation has written its result. When this restriction is not respected, the code will execute incorrectly when an operation with a multicycle latency is bisected by joining or splitting traces. This restriction implies

that the live ranges of the operands and results of the same operation always interfere when the operation is scheduled for execution on an FU with multiple pipeline stages (e.g. the latency is larger than one). When the pipeline is fully utilized, $2(d-1)$ extra registers are required for an FU pipeline of d stages. This really becomes a bottleneck on processors that can execute several multi-cycle operations in parallel, which is not uncommon.

The method as proposed by Ellis does not include the insertion of state preserve code and spill code. However, a remark is made that an operation-based list scheduler probably outperforms the instruction-based list scheduler in the context of spilling. The operation-based list scheduler can look back into the already generated schedule and can schedule spill code as early as possible. The list scheduler is always constrained to schedule newly generated code after or in the current instruction being scheduled.

No heuristics were presented to prevent the scheduler from being too greedy. Consequently, it will easily over utilize the available registers. Another potential performance bottleneck is the amount of inserted repair code. No measurements are provided to evaluate the performance of this approach.

Trace Scheduling as a Global Register Allocation Framework [FR91]
Freudenberger and Ruttenberg [FR91] observe that often registers are the most critical instruction scheduling resource. To manage them well, they describe how global register assignment is integrated into trace scheduling in the MultiFlow compiler [L$^+$93, SS93]. The scheduler drives the register assignment process to place the variables referenced within the heavily-used traces in registers. The article does not discuss the assignment of registers to variables inside the traces[1], but merely presents the communication required to keep an assignment consistent between traces. Since traces have multiple entry and exit points, repair code is inserted to obtain correct programs. When other, less crucial, traces *hook up* to this trace, extra analysis is needed to check whether a variable is allocated in the same register in all traces. When this is not true, extra code is inserted to make the necessary corrections.

Freudenberger and Ruttenberg observed that repair code for register assignment purposes alone, already contributed 5% to the total operation count. They compared their results to two other processors (with other compilers) by counting the executed operations. It is shown that the proposed method is competitive with both other approaches. However, comparing compilers from other vendors on other architectures is difficult; results cannot be generalized without listing the algorithms used by the other compilers.

Instruction Scheduling for TriMedia [HA99]
In [HA99], Hoogerbrugge and Augusteijn describe the compiler for the TriMedia VLIW mediaprocessor family. The operation-based list scheduler operates on decision trees. In practice, decision trees are often too small to contain sufficient ILP, especially in control intensive applications. Grafting is used to

[1]The assignment inside traces is based on the work of Ellis [Ell86]

remove decision tree boundaries by duplication join-point basic blocks. In order to support speculative execution, guards are assigned to operations when required. The register allocator is split into two parts: a global and a local register allocator. To support this division, the registers are divided into local and global registers. The global variables are assigned using a graph coloring based algorithm prior to instruction scheduling. A local integrated register allocator assigns the variables local to a decision tree. A register is assigned to a variable as soon as the definition is scheduled. Since the scheduler uses decision trees as a scheduling scope, all live ranges are tree-shaped (a single definition per live range) and definitions are scheduled before their uses are scheduled. This greatly simplifies integrated local register assignment.

To keep the register pressure under control, heuristics are used. Hoogerbrugge introduces the notion of *floater* operations. A floater operation has either none or a single predecessor (which is also a floater) in the DDG and its result is used only once. These operations are called floaters because they tend to float to the top of the decision tree when the list scheduler schedules them as soon as possible. This results in long live ranges and thus in an increased register pressure. Therefore, floaters are handled differently in the TriMedia scheduler. First, they are not in the *ready* set. When a non-floater is scheduled, its preceding floaters, if any, are scheduled as close as possible before it. Unfortunately, no results are given to verify the impact of this idea on performance.

When the register allocator runs out of registers, variables are spilled. Because a register is required between the operation and the actual spill and reload operations, a few registers are reserved. Without these registers the scheduler might get stuck. The insertion of state preserving code is not required since the compiler does not support procedure calls.

The proposed method divides the register set in three sets: global registers, local registers and spill registers. This can lead to an inefficient usage of registers: one set might be under utilized while the other is over utilized. This will, however, not be a severe problem for the TriMedia processor family since it contains 128 registers. For processors with a smaller amount of registers, it is expected to become a problem. Global register assignment is performed prior to instruction scheduling; this may lead to the introduction of false dependences, which limits the performance. Unfortunately, no comparison is made with a conventional approach, such as early assignment, to evaluate the presented method.

5.4 Conclusion

Register assignment and instruction scheduling are antagonistic phases in compilers that exploit ILP. The phase executed first may hinder the other. In theory, if there are an infinite number of resources, early, late and integrated assignment, generate code with the same performance. However, for register

accesses to be fast, the size of the register file should be limited. Hence, the question is: how to use a limited set of registers efficiently? This problem was addressed in this chapter. We discussed the problems related to early and late register assignment and TTA related issues. In summary:

- The assignment of registers to variables prior to instruction scheduling may limit the possibilities to reorder operations because of false dependences introduced with the re-use of registers. Lately, some work is done on the interaction between instruction scheduling and register assignment. In order to avoid the introduction of false dependences, the register allocator is made aware of the code motions the instruction scheduler wants to perform.
- Instruction scheduling uninhibited by constraints imposed by register assignment leads to efficient schedules. Unfortunately, it may also increase the span of live ranges, which leads to excessive spilling. An efficient schedule can lose its achieved degree of ILP when spill code is inserted afterwards. Heuristics are introduced that limit the greediness of the instruction scheduler.
- Early assignment cannot exploit the software bypass and dead-result move elimination advantage of TTAs. Consequently, the resulting code has a lower efficiency caused by wasted registers.
- The insertion of spill code in TTA code when using late assignment is much more complicated than for conventional architectures. This is caused by the software bypassing and dead-result move elimination capability. Furthermore, because operation latencies are visible, the insertion of instructions in already scheduled code may violate the constraints under which the code originally was scheduled.

Today, the problems related to early and late assignment hinder the generation of high performance code. At the same time ILP compiler techniques are advancing and the available silicon space increases. Both advances allow the execution of an increasing number of operations in parallel to boost performance. The more simultaneously issued operations, the more registers are potentially required. Thus, register assignment and instruction scheduling must be addressed simultaneously in order to maximize ILP.

Integrated Assignment and Local Scheduling

6

The goal of register assignment is to map the variables of a program as efficiently as possible to the set of registers of a processor to obtain fast programs and to minimize the number of executed memory accesses. The task of the instruction scheduler is to order the instructions in such a way that the execution time of a program is minimized. Both the instruction scheduler and the register allocator have the same goal: minimizing the execution time of a program. However, decisions made by one phase can deteriorate the overall performance because they put too many constraints on the other phase.

This chapter describes a global register assignment method integrated within a local operation-based list scheduler. To the best of our knowledge, no integrated approach towards global register assignment and instruction scheduling exists using an operation-based list scheduler. The method is described in the context of a basic block scheduler. The next two chapters will discuss extensions of this method for two more aggressive scheduling techniques, which are region scheduling and software pipelining.

To make a new register assignment approach applicable for use in production compilers it should incorporate all aspects of register assignment, including spilling and the insertion of state preserving code. Unlike many other researched methods, our integrated method incorporates all these aspects. Therefore, we are able to compile any ANSI C/C++ program including SPECint95 benchmarks.

This chapter is structured as follows. Section 6.1 discusses issues related to resource assignment and instruction scheduling. The register assigned to a particular variable is selected from a set of free registers. An important data structure to compute this set is described in Section 6.2. The definition of the

set itself is given in Section 6.3. When insufficient registers are available to hold all variables, spill code is inserted. The insertion of spill code is discussed in Section 6.4. In contrast to other approaches [HA99], our method allows procedure calls. The insertion of code to preserve the state of the program across procedure calls is described in Section 6.5. In Section 6.6, experiments are described to evaluate the new method. The developed integrated assignment method is implemented in the same compiler as the early and late assignment methods. This gives the opportunity to make a fair comparison with early and late assignment. Finally, Section 6.7 states the conclusions.

6.1 Resource Assignment and Phase Integration

Integration of instruction scheduling and register assignment has as a goal to generate code that is more efficient by letting the two phases interact. This complicates register assignment, because variables that were not live simultaneously before a scheduling step can be simultaneously live after this step and vice versa. In other words, the live relations between the variables can change in time during instruction scheduling. To get insight in the complexity of applying register assignment and instruction scheduling simultaneously, the impact of the assignment of other resources, such as buses and FUs, is compared with the assignment of registers.

To make correct resource assignments it is necessary to collect information about previous assignments. This is accomplished with the use of so-called *resource vectors*. For example, to record the assignment of buses to moves a *bus availability vector* is created for each instruction. When a move m is scheduled in instruction i the bus assigned to m is set as occupied in the bus availability vector associated with i. Before scheduling another move in instruction i the scheduler checks the bus availability vector for available buses. The same kind of administration is used for sockets. In terms of register assignment, the live ranges of buses and sockets always span a single instruction.

Assigning an FU to an operation involves checking the availability of this FU and, when the FU is selected, updating the appropriate resource vectors[1]. In contrast to buses and sockets, the resource vectors of an FU span multiple instructions, ranging from the instruction where the operand move is scheduled to the instruction in which the result move is scheduled. Typically, the number of spanned instructions is equal to the latency of the FU performing the operation. In other words, the live range of an FU is equal to the latency of the FU.

Variables are in general referenced by many operations. Once a variable is assigned to a register, this register cannot be used for storing other values until its content is killed. In contrast to other resources, the live range of a register

[1]The precise administration of the availability of FUs is outside the scope of this thesis. The interested reader is referred to [Hoo96].

spans many instructions and may even cross basic block and region boundaries. Therefore, registers can be considered as a global kind of resource, reserved and released at different points of the program. All other resources can be considered as local resources since these resources are reserved and released within a single or a few instructions. The larger scope of register assignment makes registers the hardest allocatable resource.

The question arises when to assign registers to variables. One approach is to assign a register to a variable v when all references to v are scheduled. At this point, all instructions spanned by the live range are known. The main reason not to choose this approach are the problems related to the insertion of spill code in already scheduled code. Therefore, a method that avoids the insertion of spill code in already scheduled instructions is chosen. The fundamental idea of our approach is as follows:

> *A register r is assigned to a variable v as soon as a move m is scheduled that refers to v.*

The complete live range of a variable is checked for a common available register, before any of the references to this variable are scheduled.

Algorithm 6.1 assigns the transport resources (buses, sockets and registers). A valid transport resource combination for a move m in instruction i consists of a source socket si, a move bus mb and a destination socket di, which are not already in use (in other words available) in this instruction. This combination should form a data path from the source register (FU or RF) to the destination register (FU or RF) in the TTA processor.

6.2 Register Resource Vectors

A register is assigned to a variable when the first move referring to this variable is scheduled. Bookkeeping is necessary to guarantee that variables with overlapping live ranges are not mapped onto the same register. Just as creating a bus availability vector for each instruction, a *Register Resource Vector* (*RRV*) is associated with each instruction.

Definition 6.1 *The Register Resource Vector $RRV(i)$ is defined as the set of registers that are in use at instruction i.*

An example is given in Figure 6.1. Note that a register can be re-defined by another operation in the same instruction as where it was last used.

When the live range of a variable v spans multiple basic blocks, the register r mapped onto v is added to the RRVs of all instructions in the spanned basic blocks. Observe that a basic block has no instructions when it is not yet selected for scheduling and therefore the usage of r cannot be recorded properly. This leads to incorrect assignments. To solve this problem, initially to each basic block a single instruction, and thus a single RRV, is added. This enables the correct recording of the assignments.

Algorithm 6.1 ASSIGNTRANSPORTRESOURCES(m, i)

```
src = SOURCE(m)
dst = DESTINATION(m)
FOR EACH si ∈ AVAILABLESRCSOCKETS(src, i) DO
  FOR EACH di ∈ AVAILABLEDSTSOCKETS(dst, i) ∧ di ≠ si DO
    FOR EACH mb ∈ AVAILABLEMOVEBUSES(si, di, i) DO
      IF ISVARIABLE(src) THEN
        r_src = SELECTSRCREGISTER(m, i)
        IF r_src = ∅ THEN
          continue
        ENDIF
      ENDIF
      IF ISVARIABLE(dst) THEN
        r_dst = SELECTDSTREGISTER(m, i)
        IF r_dst = ∅ THEN
          continue
        ENDIF
      ENDIF
      IF ISVARIABLE(src) THEN
        ASSIGNREGISTER(src, r_src)
      ENDIF
      IF ISVARIABLE(dst) THEN
        ASSIGNREGISTER(dst, r_dst)
      ENDIF
      ASSIGNSOURCESOCKET(m, si)
      ASSIGNDESTINATIONSOCKET(m, si)
      ASSIGNMOVEBUS(m, mb)
      return TRUE
    ENDFOR
  ENDFOR
ENDFOR
return FALSE
```

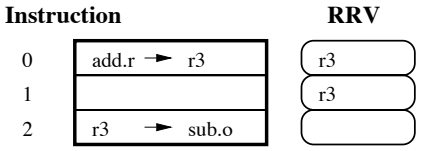
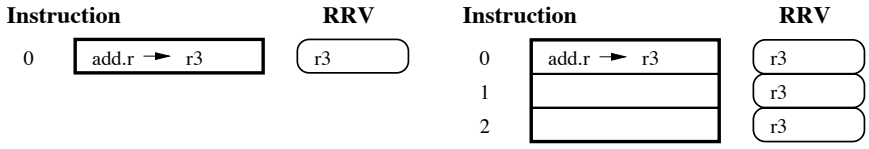


Figure 6.1: Usage of RRVs.



a) Initial schedule. b) Adding two instructions.

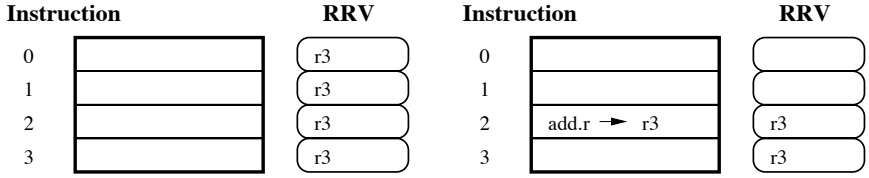
Figure 6.2: The impact on RRVs when enlarging a basic block.

During scheduling of a basic block b, new instructions are added to it when an operation cannot be scheduled in the currently available instructions. The RRVs associated with these added instructions are initialized with the information recorded in the RRV of the last instruction of b. This is valid because a register is available again in the instruction in which it is killed (the last use of the variable onto which the register is mapped) and the live information of the newly added instructions is identical to the live information of the last instruction of b. An example, which illustrates the addition of instructions, is given in Figure 6.2. Figure 6.2a shows a basic block with a scheduled definition of register $r3$. In Figure 6.2b, the basic block is enlarged with two instructions. The contents of the last RRV is copied, hence $r3$ is also set to be unavailable in the newly added instructions.

When a register is selected and assigned to a variable v, the RRVs must be updated to guarantee that subsequent assignments are legal and do not interfere with this assignment. Each time a move referring to variable v is scheduled, more information about the size of its live range is known. As a result, the RRV information can be refined.

A register is assigned to v when the first move referring to it is scheduled. Consequently, all the *other* moves referring to v have already a register assigned to them, before they are scheduled. When such a move, for example a definition is scheduled, the register associated with it can be set as available again in the instructions before this definition. This is illustrated in Figure 6.3. Figure 6.3a shows the situation before the definition is scheduled. In Figure 6.3b the definition is scheduled and the appropriate RRVs are updated. There is, however, one exception. When a use of a register was already scheduled in a lower or the same instruction as the definition, the register can only be removed in the instructions between this use and the definition.

The RRVs are also updated when a use is scheduled, to which already a register was assigned. Figures 6.4b to Figures 6.4e show various scenarios when scheduling the code fragment of Figure 6.4a, it is assumed that `r3` is not in $live_{Out}$. Figure 6.4b shows the situation when the definition and the first use are scheduled. Since it is not known in which instruction the second use is going to be scheduled the worst is assumed and `r3` is not available in all instructions of the basic block. Figure 6.4c shows the contents of the RRVs when the



a) RRVs prior to scheduling the definition.

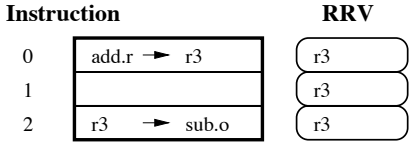
b) RRVs when the definition is scheduled.

Figure 6.3: Updating RRVs after scheduling a definition.

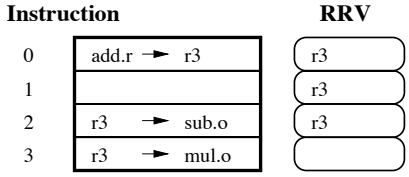
second use is also scheduled. More information about the live range is known and the RRVs are updated as shown in the figure. Since operation-based list scheduling is used, the second use can be scheduled in an earlier instruction than the first use. In this situation, in the second instruction, register `r3` can be released for re-usage. This is shown in Figure 6.4d. Not all live ranges are local to a basic block. Assume, in our example, that the second use of `r3` is located in a successor basic block. The RRVs associated with the instructions of the successor basic block must all contain register `r3`. When the second use is scheduled, the RRVs in the successor basic block can be updated. This situ-

```
add.r → r3;
r3 → sub.o;
r3 → mul.o;
```

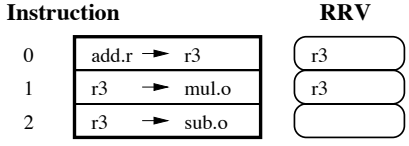
a) Code fragment.



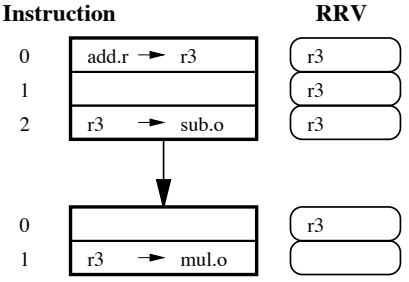
b) Definition and first use scheduled.



c) Second use scheduled after first use.

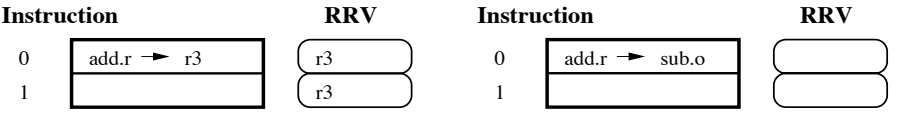


d) Second use scheduled before first use.



e) Variable live across basic block boundaries.

Figure 6.4: Updating RRVs after scheduling a use.



a) Initial schedule.

b) Software bypassing and dead-result move elimination.

Figure 6.5: Updating RRVs when applying software bypassing and dead-result move elimination.

ation is depicted in Figure 6.4e. Because register `r3` is live until the end of the original basic block, it is not released in its last RRVs.

TTAs have a property that makes integrated register assignment even more attractive: its ability to forward data directly from the output of one FU to the input of another or the same FU. How this advantage is exploited in our integrated assignment method is shown in Figure 6.5. Figure 6.5a gives the situation where a definition of register `r3` is scheduled. Figure 6.5b shows the schedule and its RRVs when the variable is software bypassed, and this use ends the live range (i.e. dead-result move elimination can be applied). The variable disappears from the code and therefore also register `r3` is removed from the RRVs and can be used again for another variable. The register can only be removed from the RRVs when all uses are scheduled in the same instruction as their definition.

6.3 The Interference Register Set

The *interference register set* of a variable v contains the registers that are mapped onto variables that may interfere with v under any legitimate schedule. This set is used for the selection of a register for variable v. For each basic block b in the live range of v, an interference register set is constructed.

The basic blocks spanned by the live range of variable v can be partitioned into two sets.

- $B_{IO}(v)$, variable v is live on entry and exit of these basic blocks, but it is not referenced. This set is constructed using[2]:

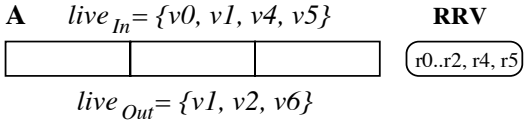
$$B_{IO}(v) = \{b \in B | v \in live_{In}(b) \wedge v \in live_{Out}(b) \wedge v \notin live_{Use}(b)\} \quad (6.1)$$

  All variables live in these basic blocks interfere with variable v. The interference register set $R_{IO}(v, b)$ for a basic block $b \in B_{IO}(v)$ is simply

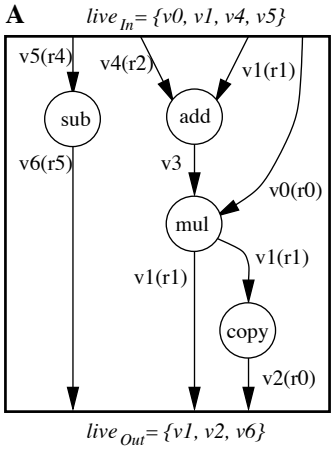
[2]Note that due to the definitions of the sets $live_{In}$, $live_{Out}$, $live_{Def}$ and $live_{Use}$ it is not necessary to include $v \notin live_{Def}(b)$ in the construction of this set.

```
v4(r2) → add.o; v1(r1) → add.t;
add.r → v3;
v3 → mul.o;     v0(r0) → mul.t;
mul.r → v1(r1);
v1(r1) → v2(r0);
v5(r4) → sub.o; #4 → sub.t;
sub.r → v6(r5);
```

a) Unscheduled TTA code.



b) The unscheduled basic block A with its RRV.



c) The DDG of basic block A.

Figure 6.6: RRV based register interference.

computed with:

$$R_{IO}(v,b) = \bigcup_{\forall i \in b} RRV(i),\ b \in B_{IO}(v) \tag{6.2}$$

- $B_{DU}(v)$, these basic blocks have references to variable v. This set is described with:

$$B_{DU}(v) = \{b \in B \mid v \in live_{Def}(b) \vee v \in live_{Use}(b)\} \tag{6.3}$$

The set of interfering registers of a basic block $b \in B_{DU}(v)$ is denoted with $R_{DU}(v,b)$.

Computing the interference register set $R_{DU}(v,b)$ is much more complicated than computing $R_{IO}(v,b)$. A conservative approach is simply to include all registers in the RRVs of all instructions of the basic blocks $b \in B_{DU}(v)$. Figure 6.6 gives an example. Figure 6.6a shows the operations of a basic block **A**. The notation `v4(r2)` means register `r2` is mapped onto variable `v4`. Figure 6.6b gives the (empty) schedule and the RRV. According to the RRV information variable `v3` cannot be mapped onto any of the registers `r0`, `r1`, `r2`, `r4` and `r5`. However, careful examination of the code learns that registers `r1` and `r2` never interfere with variable `v3`, independent of the generated schedule. Consequently, the information in the RRV is too conservative.

To efficiently exploit the available registers, a more accurate estimation is required. The DDG's partial ordering within basic blocks is used for constructing the interference set. To capture all interference types, four non-interference sets are defined per basic block. In the formulas, all references to variables are made

by operations belonging to basic block b, i.e., $n_{use(v)}, n_{use(v_i)}, n_{def(v)}, n_{def(v_i)}$ are contained in b.

Definition 6.2 *The non-interference set $V_{Below}(b, v)$ contains all variables v_i, whose live range starts after the end of the live range of v in basic block b. The ordering of the live ranges is not the result of the ordering in the sequential intermediate code, but is the result of the dependence relations in the DDG. The set $V_{Below}(b, v)$ is constructed with:*

$$V_{Below}(v, b) = \left\{ v_i \in live_{Def}(b) \mid (n_{use(v)}, n_{def(v_i)}) \in E^T \; \forall \, n_{use(v)} \in b \right\} \quad (6.4)$$

where E^T is the set of edges of the transitive closure of the DDG.

A similar situation arises when v and v_i change roles.

Definition 6.3 *The non-interference set $V_{Above}(b, v)$ contains all variables v_i, whose live range ends before the live range of v starts in basic block b. More formally:*

$$V_{Above}(v, b) = \left\{ v_i \in live_{\neg Out}(b) \mid (n_{use(v_i)}, n_{def(v)}) \in E^T \; \forall \, n_{use(v_i)} \in b \right\} \quad (6.5)$$

where $live_{\neg Out}(b) = (live_{Def}(b) \cup live_{In}(b)) - live_{Out}(b)$.

Things become more complex when the live range of v_i is loop carried, e.g. the variable v_i is live at entry of basic block b and it is redefined within b.

Definition 6.4 *The non-interference set $V_{Around}(v, b)$ contains all variables that do not interfere with v in basic block b, and are live on entry and are redefined in basic block b. This set is constructed with:*

$$\begin{aligned} V_{Around}(v, b) = \{ \; & v_i \in live_{Loop}(b) \mid (n_{use(v)}, n_{def(v_i)}) \in E^T \; \forall \, n_{use(v)} \in b \\ & \wedge \; (n_{use(v_i)}, n_{def(v)}) \in E^T \; \forall \, n_{use(v_i)} \in K(v_i, b) \; \} \end{aligned} \quad (6.6)$$

where $live_{Loop}(b) = live_{In}(b) \cap live_{Def}(b)$ and $K(v, b)$ is the set of uses of v, which will be executed before the definition of v in basic block b.

$$K(v, b) = \left\{ n \in N_{Use(v)} \mid (n_{def(v)}, n_{use(v)}) \notin E^T, n_{def(v)}, n_{use(v)} \in b \right\} \quad (6.7)$$

A similar situation occurs when the roles of v and v_i are interchanged.

Definition 6.5 *The non-interference set $V_{Between}(v, b)$ contains all variables that do not interfere with v in basic block b, when v is live on entry and exit of b. More formally:*

$$\begin{aligned} V_{Between}(v, b) = \{ & v_i \in live_{Local}(b) \mid (n_{use(v_i)}, n_{def(v)}) \in E^T \; \forall \, n_{use(v_i)} \in b \\ & \wedge \; (n_{use(v)}, n_{def(v_i)}) \in E^T \; \forall \, n_{use(v)} \in K(v, b) \; \} \end{aligned} \quad (6.8)$$

where $live_{Local}(b) = live_{Def}(b) - live_{Out}(b)$.

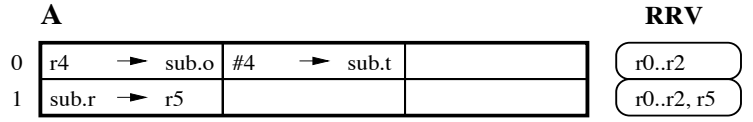


Figure 6.7: Partial schedule of basic block **A**.

The non-interference set of a basic block $b \in B_{DU}(v)$ can now be computed with the four non-interference sets of the Equations 6.4, 6.5, 6.6 and 6.8:

$$V_{\textit{non-interf}}(v,b) = V_{\textit{Below}}(v,b) \cup V_{\textit{Above}}(v,b) \cup V_{\textit{Around}}(v,b) \cup V_{\textit{Between}}(v,b) \qquad (6.9)$$

The set of interfering registers in $b \in B_{DU}(v)$ can now be determined with:

$$R_{DU}(v,b) = \{r(v_i) \mid v_i \in \mathit{live}(b) - V_{\textit{non-interf}}(v,b) - v\} \qquad (6.10)$$

where $r(v_i)$ returns the register mapped on variable v_i. When no register is assigned to v_i then $r(v_i) = \emptyset$.

Figure 6.6c shows the portion of the DDG of the basic block of Figure 6.6a. The figure only shows a small part of the DDG of the complete procedure, which explains the dangling edges. The following sets are now constructed: $V_{\textit{Below}}(\texttt{v3}, \mathbf{A}) = \{\texttt{v2}\}$, $V_{\textit{Above}}(\texttt{v3}, \mathbf{A}) = \{\texttt{v4}\}$, $V_{\textit{Around}}(\texttt{v3}, \mathbf{A}) = \{\texttt{v1}\}$ and $V_{\textit{Between}}(\texttt{v3}, \mathbf{A}) = \emptyset$. According to Equation 6.9, the set of non-interfering variables becomes $V_{\textit{non-interf}}(\texttt{v3}, \mathbf{A}) = \{\texttt{v1}, \texttt{v2}, \texttt{v4}\}$. Since $\mathit{live}(\mathbf{A}) = \{\texttt{v0}, \texttt{v1}, \texttt{v2}, \texttt{v3}, \texttt{v4}, \texttt{v5}, \texttt{v6}\}$ the set of interfering registers becomes $R_{DU}(\texttt{v3}, \mathbf{A}) = \{\texttt{r0}, \texttt{r4}, \texttt{r5}\}$. The registers in the set $\{\texttt{r0}, \texttt{r4}, \texttt{r5}\}$ may interfere with variable $\texttt{v3}$ under any legitimate schedule.

For all basic blocks in the set $R_{DU}(v,b)$, the interference register set is computed using Equation 6.10. These basic blocks are not scheduled yet. There is, however, one exception: the currently scheduled basic block b^*. The set $R_{DU}(v,b^*)$ indeed contains all registers that might possibly interfere with v, prior to scheduling any of the operations of b^*. However, when the scheduler has already assigned some operations to instructions, some of the registers do not interfere anymore. This is illustrated in Figure 6.7, which shows the schedule of our running example (see Figure 6.6) after scheduling the subtraction. As can be seen, variable $\texttt{v3}$ can never interfere anymore with register $\texttt{r4}$ because the definition of $\texttt{v3}$ will always be scheduled after the use of $\texttt{r4}$. When the information in the RRVs is combined with $R_{DU}(v,b^*)$, a more accurate interference set can be constructed. The exact construction of this set depends on whether a definition or a use of a variable v is scheduled.

- When a definition n of variable v is scheduled in instruction i_{cur}, only the RRVs of instruction i_{cur} until the last instruction of basic block b^* need to

be checked for a free register.

$$R_{RRV}(v,n) = \bigcup_{i=i_{cur}}^{LastInsn(bb(n))} RRV(i) \tag{6.11}$$

- A similar approach is used for constructing the interference register set when a use n of variable v is scheduled in instruction i_{cur}.

$$R_{RRV}(v,n) = \bigcup_{i=0}^{LastUseInsn(n,v)} RRV(i) \tag{6.12}$$

where

$$LastUseInsn(n,v) = \begin{cases} LastInsn(bb(n)) & : \; N_{Use(v)} - n \neq \emptyset \\ LastInsn(bb(n)) & : \; v \in live_{Out}(bb(n)) \\ i_{cur} - 1 & : \; otherwise \end{cases} \tag{6.13}$$

Combining the sets $R_{RRV}(v,n)$ and $R_{DU}(v,b^*)$ results in an instruction precise registers interference set of variable v, for any possible code ordering of the *remaining* unscheduled operations in basic block b^*. This set, $R_{Cur}(v,n)$, is computed with:

$$R_{Cur}(v,n) = R_{RRV}(v,n) \cap R_{DU}(v, bb(n)) \tag{6.14}$$

The first scheduled move in our running example (see Figure 6.6 and 6.7), referring to `v3`, is a definition (e.g., `add.r` $\rightarrow$ `v3`). The first instruction in which it can be scheduled is instruction 1. As a result $R_{RRV}(\texttt{v3}, \texttt{add.r} \rightarrow \texttt{v3}) = \{\texttt{r0}, \texttt{r1}, \texttt{r2}, \texttt{r5}\}$ and $R_{Cur}(\texttt{v3}, \texttt{add.r} \rightarrow \texttt{v3}) = \{\texttt{r0}, \texttt{r5}\}$.

The complete set of interfering registers can now be computed with:

$$\begin{aligned} R_{\mathit{Interfere}}(v,n) \quad = \quad & \left(\bigcup_{b \in B_{IO}(v)} R_{IO}(v,b) \right) \cup R_{Cur}(v,n) \\ & \cup \left(\bigcup_{b \in B_{DU}(v) - bb(n)} R_{DU}(v,b) \right) \end{aligned} \tag{6.15}$$

6.4 Spilling

In this section, issues related to spilling in the context of our integrated assignment method are discussed. Late and early assignment insert spill code in either completely scheduled code, or completely unscheduled code. Integrated assignment has to insert spill and reload code in *partly scheduled* code. In Section 6.4.1, a solution is presented, which solves this problem. Adding

operations changes the data dependence relations and the data flow relations. This issue is addressed in Section 6.4.2. Scheduling of on-the-fly inserted spill code has complications. These complications are identified in Section 6.4.3 and their solutions are presented.

6.4.1 Integrated Spilling

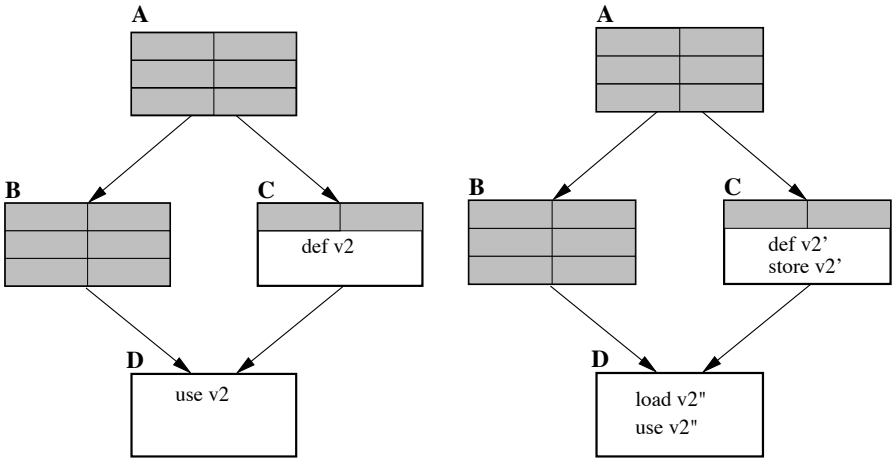
The problem of inserting spill code in partly scheduled code seems to be similar to the problem of inserting spill code in completely scheduled code. Extra operations must be squeezed into scheduled instructions. As already discussed in Section 5.2.3, this requires rescheduling in order to generate correct code. Rescheduling may lead to changes in the register requirements; non-interfering live ranges in the original scheduled code may interfere in the rescheduled code, or variables that were software bypassed, are not software bypassed anymore, and require registers. These effects result in an iteration of register assignment, spill code insertion and rescheduling steps. It is unclear how to apply this strategy in the context of integrated assignment because it is not desirable to restart scheduling, when during scheduling it is discovered that spilling is required.

Because of the above mentioned reasons, it was decided not to insert spill code in already scheduled code. Instead, spill code is only inserted in the still to be scheduled code. This strategy fits very well in our integrated assignment approach, because a register is assigned to a variable when the first reference to this variable is being scheduled. As a result, all references are located in unscheduled basic blocks. This avoids the insertion of code in already scheduled code and therefore is easier to implement.

The principle of our approach is illustrated in the example of Figure 6.8. Figure 6.8a shows the CFG of a procedure. Assume that the basic blocks **A** and **B** are already scheduled and basic block **C** is being scheduled. The shaded parts indicate which code is already scheduled. In this example, it is assumed that no more free registers are available in basic block **D**. Consequently, no register can be found for the live range of variable `v2`. Integrated assignment detects this situation when it tries to schedule the definition of `v2` in basic block **C**. As illustrated in Figure 6.8b spill and reload code is inserted in the unscheduled code of respectively basic block **C** and **D**. For reasons of clarity the code for the address calculations is omitted.

6.4.2 Updating Data Flow and Data Dependence Relations

Spill code insertion changes the data dependence and data flow relations. This information was originally computed before scheduling, now, during scheduling the necessary updates must be made to ensure correct code generation. New live ranges are created to hold the memory addresses, and the to be spilled and reloaded values. To maintain the fully renaming property, and



a) Before spilling. b) After Spilling.

Figure 6.8: Integrated spilling.

thus a large scheduling freedom, new and *unique* variable names are associated with these live ranges. In the following, the changes to the DDG are described in detail when spilling a variable v.

- A store operation st_{def_i} is inserted just after each operation $n_{def_i(v)} \in N_{Def(v)}$. A unique index i is given to each definition of v in $N_{Def(v)}$. A data dependence edge, of the flow dependence type, is added between the definition $n_{def_i(v)}$ and the associated st_{def_i}.

$$\begin{aligned} N_{DDG} &= N_{DDG} \cup \{st_{def_i}\} \\ E_{DDG} &= E_{DDG} \cup \left\{(n_{def_i(v)}\ \delta_0^f\ st_{def_i}) \mid st_{def_i}, n_{def_i(v)} \in N_{DDG}\right\} \end{aligned}$$

  An addition add_{def_i} is inserted just before each inserted st_{def_i}. This addition computes the memory address of the location where the spilled variable is stored. The DDG is updated with:

$$\begin{aligned} N_{DDG} &= N_{DDG} \cup \{add_{def_i}\} \\ E_{DDG} &= E_{DDG} \cup \left\{(add_{def_i}\ \delta_0^f\ st_{def_i}) \mid add_{def_i}, st_{def_i} \in N_{DDG}\right\} \end{aligned}$$

- A load operation ld_{use_j} is inserted just before each operation $n_{use_j(v)} \in N_{Use(v)}$. The index j distinguishes the various uses of v in $N_{Use(v)}$. A data dependence edge is added between the load and the related consumer:

$$\begin{aligned} N_{DDG} &= N_{DDG} \cup \left\{ld_{use_j}\right\} \\ E_{DDG} &= E_{DDG} \cup \left\{(ld_{use_j}\ \delta_0^f\ n_{use_j(v)}) \mid ld_{use_j}, n_{use_j(v)} \in N_{DDG}\right\} \end{aligned}$$

An addition add_{use_j} is inserted just before each inserted ld_{use_j}. This addition computes the memory address of the location from which the spilled variable should be reloaded.

$$\begin{aligned} N_{DDG} &= N_{DDG} \cup \{add_{use_j}\} \\ E_{DDG} &= E_{DDG} \cup \left\{(add_{use_j}\ \delta_0^f\ ld_{use_j}) \mid add_{use_j}, ld_{use_j} \in N_{DDG}\right\} \end{aligned}$$

- Two types of memory data dependence edges are added between the inserted store and load operations. The first edge prevents that a value is read from memory before it is written. The flow dependence edge between $n_{def_i(v)}$ and $n_{use_j(v)}$ is replaced with a memory flow dependence edge between the st_{def_i} and ld_{use_j}.

$$\begin{aligned} E_{DDG} = {} & E_{DDG} \cup \left\{(st_{def_i}\ \delta_1^f\ ld_{use_j}) \mid (n_{def_i(v)}\ \delta_0^f\ n_{use_j(v)}) \in E_{DDG}\right\} \\ & - \left\{(n_{def_i(v)}\ \delta_0^f\ n_{use_j(v)})\right\} \end{aligned}$$

  The second memory dependence edge prevents that a value is written to memory before it is read. This edge replaces the anti dependence edge between $n_{use_j(v)}$ and $n_{def_i(v)}$. Such an edge only exists when variable v was loop carried.

$$\begin{aligned} E_{DDG} = {} & E_{DDG} \cup \{(ld_{use_j}\ \delta_1^a\ st_{def_i(v)}) \mid (n_{use_j(v)}\ \delta_0^a\ n_{def_i(v)}) \in E_{DDG}\} \\ & - \{(n_{use_j(v)}\ \delta_0^a\ n_{def_i(v)})\} \end{aligned}$$

The sets with live information are also updated. Variable v is removed from all the sets $live_{In}$, $live_{Out}$, $live_{Def}$ and $live_{Use}$. The new variables, created to hold the temporary values, are added to the $live_{Def}$ sets of their basic blocks. In addition, for each new live range a new du-chain is created[3].

After the insertion of spill code, the operation sequences for spilling and reloading are stand-alone pieces of code. That is, they are no longer directly connected by a du-chain and the data is transported via memory.

6.4.3 Scheduling Issues

Normally, an operation is scheduled in the first instruction where its data dependence and resource constraints are met. The used basic block scheduling method guarantees that there always exists an instruction in which both the data dependence and the local resource constraints (FUs, buses and sockets) can be fulfilled. When necessary, new (empty) instructions are created and

[3]The additions use the frame-pointer (`fp`) to compute the address of a memory location. In order to be complete, du-chains and data dependency edges between the definition of the frame-pointer and the uses of it by the additions must be inserted. For reasons of clarity, they are omitted in the above discussion.

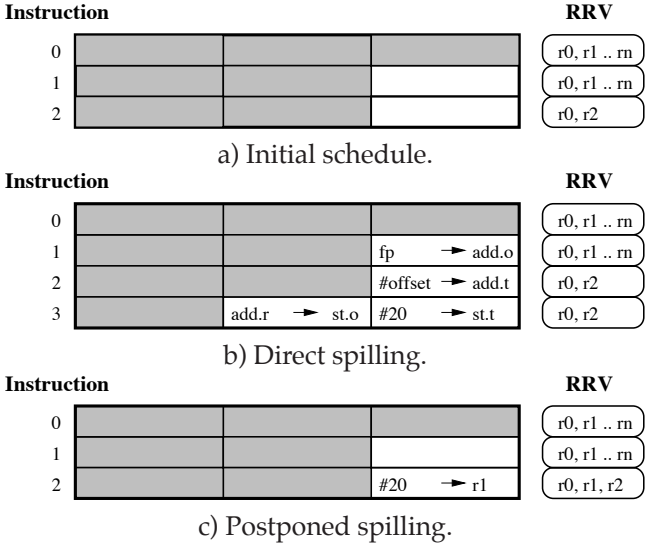


Figure 6.9: Direct vs. postponed spilling when scheduling transport `#20` → `v1`.

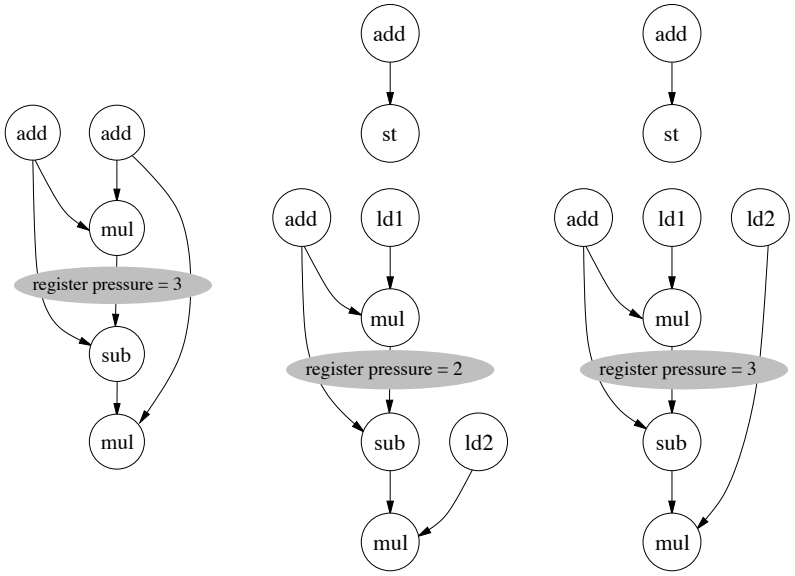
added to the basic block. For global resources, such as registers, it is *not* guaranteed that the resource constraints can be met, because variables can cross basic block boundaries.

Two strategies were explored when the inability to schedule an operation in a particular instruction is caused by register shortage:

- *Direct spilling:* Spill code is generated in the first instruction where all data dependence and all resource constraints in an instruction can be met, except registers.
- *Postponed spilling:* The scheduler tries to schedule the transport in later instructions, hoping that in one of these instructions more registers become available. When no extra registers become available, the first strategy is used as a fallback strategy.

Figure 6.9 shows the impact of both strategies. Transport `#20` → `v1` must be scheduled in the basic block given in Figure 6.9a. Direct spilling inserts spill code because all resource and dependence constraints, except registers, are met in instruction 1. This is shown in Figure 6.9b. When postponed spilling is used (see Figure 6.9c), the scheduler discovers that a register becomes available when the transport is scheduled in instruction 2. Consequently, no spill code is inserted.

Preliminary experiments indicated that postponed spilling results in a higher performance. Despite the engineering complexities, this strategy is chosen for implementation in our integrated assignment approach.



a) Original DDG. b) DDG with spill code. c) Scheduled code.

Figure 6.10: Impact of spilling and instruction scheduling on register pressure.

The idea behind spilling is to reduce the register pressure by replacing a long live range with a number of short live ranges. However, in some situations spilling increases the register pressure. For example, when inserting a store operation, the original live range is replaced with two simultaneously live, short live ranges: one for the memory address calculation and one for the value to be stored in memory. This is a problem when insufficient registers are available to hold these short live ranges; a condition very probable since spilling is due to register shortage. Other register pressure problems are related to instruction scheduling. The instruction scheduler may decide to schedule the operations required for spilling far apart. This also increases register pressure, see for instance Figure 6.10. In Figure 6.10a the original graph is shown, this program requires three registers. When only two registers are available, spill code is introduced as shown in Figure 6.10b. The resulting code requires only two registers. Because instruction scheduling techniques tend to schedule instructions as early as possible, it can happen that operation `ld2` is scheduled in the same instruction as operation `ld1`, see Figure 6.10c. In this situation, the register requirement is still three, and spilling did not help at all.

In [LVA96] this problem is attacked by scheduling the operation, which variable is spilled, and the spill code itself close together in a single scheduling step. However, this does not guarantee that no deadlock situation can arise, since there are still registers required for the introduced short live ranges.

In [CLM+95, HA99], it is suggested to use a limited set of reserved registers for these newly created live ranges. This method has three major drawbacks:

```
fp → add.o;  #offset → add.t;
add.r → ld.t;
ld.r → use;
```

Figure 6.11: Software bypassed reload code.

(1) variables are spilled to memory, although some registers were available, (2) false dependences are introduced that could be avoided if the complete set of registers was available, and (3) it introduces false dependences between spill code because the newly created live ranges are only mapped onto a small set of registers. Consequently, reserving registers for the short live ranges introduced by spilling results in inefficient register usage.

A better solution is to exploit the software bypassing property of TTAs and dead-result move elimination. The newly created live ranges are directly transported from FU to FU; they disappear completely from the code. Because no registers are required, it is guaranteed that this method always converges. The corresponding TTA code for reload code is given in Figure 6.11. The result of the addition is bypassed to the load, and the result of the load is bypassed to the operation that uses the reloaded value.

To ensure that the code can be scheduled without the need of reserved registers, the address calculation, the load or store, and the operation, which requires spilling, must be scheduled in such a way that all variables are bypassed when required. This cannot be guaranteed when the involved operations are scheduled in individual scheduling steps. For example: assume the result move of the addition performing the memory address calculation is scheduled in instruction i. The trigger of the load should also be scheduled in this instruction. However, when not enough move buses or FUs are available the trigger will be scheduled in instruction $i + 1$ or higher, and software bypassing cannot be applied.

To guarantee software bypassing, the address calculation, the load or store, and the operation, which requires spilling are scheduled in a single (atomic) scheduling step. To achieve this, the scheduler recognizes spill code. It selects stand-alone pieces of spill or reload code as if it were a single operation. The scheduler uses backtracking to ensure software bypassing. The scheduler is not always required to software bypass variables. When, for example, spilling was caused by an assignment in another basic block, some registers may still be available in the currently scheduled basic block. In these situations, the scheduler is allowed to use these available registers. This decreases the number of backtracking steps. Scheduling of spill code is a complex engineering challenge, especially when another variable, defined or used by the operation that originally required spilling, also requires spilling.

The most general spill code data dependence graph is shown in Figure 6.12, intra operation edges are omitted. It shows the worst case situation for an operation op with n operands and m results. In practice most operations have only one or two operands and one result; furthermore it is not likely that all

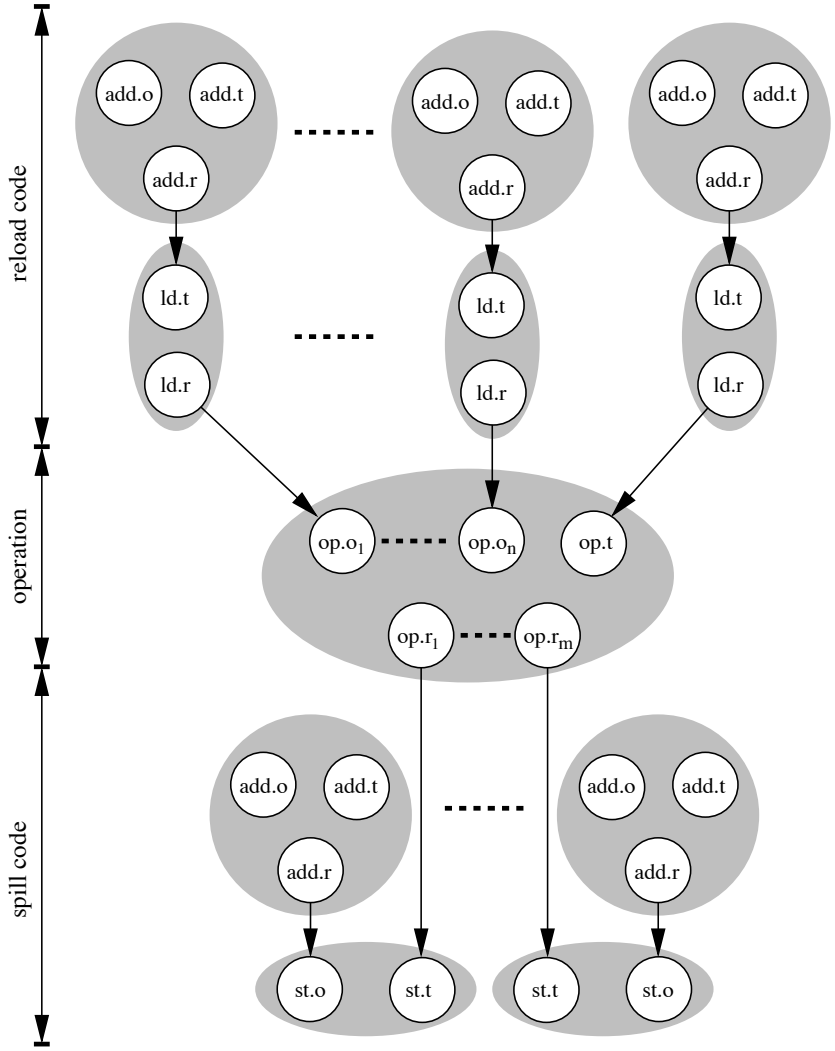


Figure 6.12: Largest recognizable spill/reload code sequence.

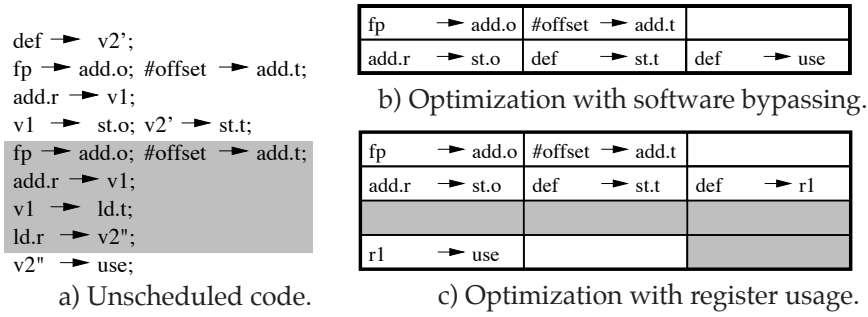


Figure 6.13: Definition-use peephole optimization.

operands and results need spill/reload code. When loads and/or stores with address offsets are supported, the graph complexity reduces substantially.

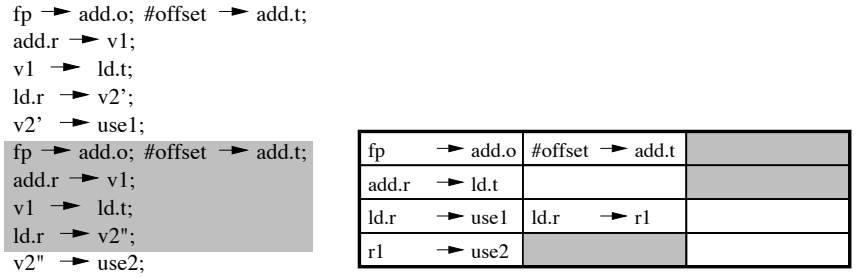
6.4.4 Peephole Optimizations

Up to now, the general spilling process is outlined, but there are cases where some of the added operations turn out to be superfluous. Integrated assignment considers several particular cases:

- When a definition and a use of a variable v are scheduled in nearby instructions, and v is spilled, the reload code can be omitted when the use can be scheduled in the same instruction as the definition. The value can be software bypassed directly to the use, without reloading the value from memory. An example is given in Figure 6.13. Figure 6.13a shows the code containing spill and reload code. The reload code (shaded in the figure) can be left out in the generated schedule. This schedule is shown in Figure 6.13b; the result of the definition (def) is spilled to memory *and*, directly software bypassed to the use.

  When a register is available between the definition and the use, this register can be used for holding the value generated by the definition. Again the reload code can be omitted. This is shown in Figure 6.13c. Register `r1` is used for temporary storage. The discarding of the superfluous operations is carried out when the use and its associate spill code is scheduled. Only at this point, it is known in which instructions the definition and the use are scheduled, and whether any registers are available.

- When two uses of the same variable are scheduled in nearby instructions, the reload code of one of the uses can be omitted. This is illustrated in Figure 6.14. Figure 6.14a shows a code fragment, which reloads the same value from memory twice. This code can be optimized by removing the second reload (shaded in the figure) and replacing it with a short live range from the first load operation to the second use. Figure 6.14b shows the resulting schedule. The discarding of the superfluous operations is carried out, when the second use and its associate spill code is scheduled.



a) Unscheduled code. b) Optimization with register usage.

Figure 6.14: Use-use peephole optimization.

6.5 State Preserving Code

When a procedure invokes another procedure, parameters are passed from the calling procedure to the called procedure, and on return from the called procedure to the calling procedure. These parameters are located in the variables `v0..v6` for integer values, and `vf0..vf4` for floating-point values, as defined by the front-end (Section 3.1). Precautions have to be taken, to ensure that the contents of these variables are not altered by register assignment. Therefore, integrated assignment assigns prior to scheduling the correct registers to these variables, just as in the graph coloring approach. These registers, however, are not dedicated for parameter passing exclusively[4]. They can also be used for holding other variables, as long as their live ranges do not interfere.

An invoked (called) procedure normally changes the contents of the registers that are in use by the calling procedure. To save the contents of these registers, state preserving code must be inserted. In the remainder of this section, methods to generate caller- and callee-saved code, in the context of integrated assignment, are discussed. Section 6.5.1 discusses the generation of callee-saved code and Section 6.5.2 describes the approach used for generating caller-saved code.

6.5.1 Generation of Callee-saved Code

The convention used in our compiler dedicates the upper half of the register set to callee-saved registers. It is the responsibility of the called procedure to save these registers when it is called. Callee-saved code is inserted in the entry and exit basic blocks of a procedure. The registers saved and restored are those which are referenced in the called procedure and are a member of the callee-saved register set. In the context of integrated assignment two methods for inserting callee-saved code are developed:

[4]The registers `sp` and `fp` can never be used by another variable, since they are live in the complete procedure.

- Create an extra hierarchy level by adding extra entry and exit basic blocks in the same way as Sweany and Beaty [SB90] propose for late assignment. These added basic blocks are dedicated to hold callee-saved code solely. This approach has as a drawback that the callee-saved code is not scheduled with the code of the original entry and exit basic blocks: this will likely result in a small performance loss.
- To overcome the limitation of the previous method the callee-saved code must be inserted within the original entry and exit basic blocks. To generate legal code the following steps should be performed:

  1. Schedule all basic blocks, except the entry and exit basic blocks.
  2. Map all remaining, not yet assigned variables whose references are located in the not yet scheduled entry and exit basic blocks, onto registers. Thus effectively applying early assignment to these basic blocks. When no register can be found, no spill code is generated in the hope that integrated assignment can find a free register during scheduling.
  3. Insert save-code in the entry basic block, and restore-code in the exit basic blocks for each referenced callee-saved register in this procedure. To ensure that the operations are scheduled in the correct order, extra data dependency edges must be added between the callee-saved code and all not yet scheduled references to the callee-saved registers.
  4. Set all never referenced callee-saved registers as used in the RRVs of the entry and exit basic blocks. For these registers, no callee-saved code is generated. This prevents the use of these registers by the yet to be scheduled operations, and thus avoids the insertion of callee-saved code in already scheduled code.
  5. Schedule the operations in the entry and exit basic blocks.

  The drawback of this method is that the variables, which were not yet mapped onto registers, can only be mapped onto the caller-saved registers and the saved callee-saved registers[5]. This generally does not impose severe problems; when there are many registers the probability of finding a register is high, since at least all caller-saved register are available. When there are only a few registers, all callee-saved registers are used in the other basic blocks and thus all registers are available.

Both methods avoid the problem of inserting callee-saved code in already scheduled basic blocks. The second method potentially results in a larger amount of exploitable ILP. It places the callee-saved code and the code of the original entry and exit basic blocks into the same basic block. Based on this

[5]When no register can be found at all, the variable is of course spilled.

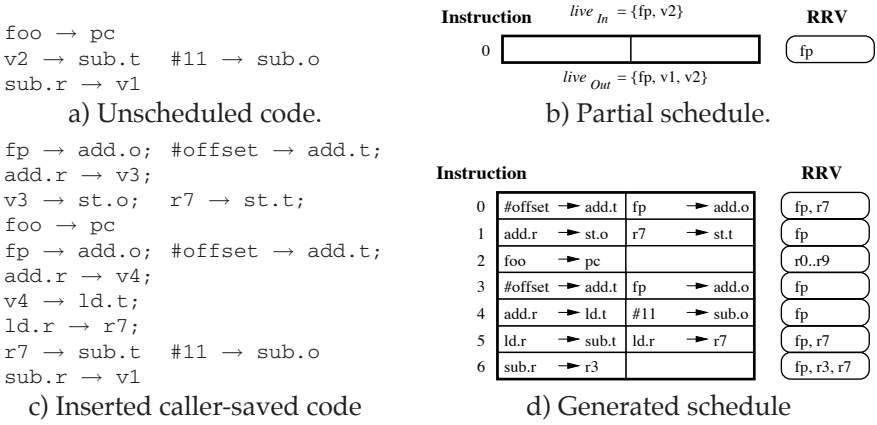


Figure 6.15: On-the-fly caller-saved code generation.

observation and preliminary experiments, the second approach is selected and incorporated within the TTA compiler back-end.

6.5.2 Generation of Caller-Saved Code

It is the responsibility of the register allocator to save and restore caller-saved registers around procedure calls with the use of caller-saved code. In an integrated assignment approach, this code cannot be generated before scheduling the procedure. At that point in time, no registers are assigned yet, and thus it is unknown which registers are alive across the procedure calls. Instead, the caller-saved code must be generated at the moment the procedure call operation is selected for scheduling. The following steps show how this problem is solved in our integrated assignment approach.

1. First, the variables that are live across the procedure call are identified by using live-variable information. Because other operations are already scheduled, it is very likely that some of these variables are already mapped onto a register. For the unassigned variables, the integrated assignment method has the freedom to map them onto either the caller- or the callee-saved register set. The same heuristic as in early assignment is used to determine the best register set (see Section 3.3.3). When a caller-saved register is selected, the variable is mapped onto this register although *no* reference to this variable is scheduled yet. An example is given in Figure 6.15. Figure 6.15a shows the unscheduled code and Figure 6.15b shows the produced schedule so far. The next operation to be scheduled is the procedure call $foo \rightarrow pc$. Prior to scheduling the procedure call, variable `v2` ,which is live across the procedure call, is mapped onto the caller-saved register `r7`.
2. Caller-saved code is inserted for all variables that are mapped onto a

caller-saved register and are live across the procedure call. In the sequential code, *save-code* is inserted *before* the procedure call, and consists of a store and an add operation. *Restore-code*, a load and an addition, is inserted *after* the procedure call in the sequential code. The additions are required to compute the memory locations. An incremental live-variable algorithm is used for generating the new live-variable information. Extra data dependences are inserted to prevent that the loads of the restore code can be scheduled before the procedure call. Figure 6.15c shows the unscheduled (sequential) code with the inserted state preserving code for register `r7`.

3. The operations of the inserted save-code are scheduled in the same manner as all other operations.
4. When all save-code operations are scheduled, the procedure call is scheduled. This implies that it is no longer allowed to assign a caller-saved register to an unassigned variable that is live across this procedure call. Without this restriction, the required extra save-code must be inserted in already scheduled code. As discussed previously, this is problematic for TTAs. To prevent such an assignment, all caller-saved registers are set as occupied in the RRV of the instruction of the procedure call. This is illustrated in Figure 6.15d assuming a TTA with 20 registers. All caller-saved registers (`r0` - `r9`) are set as occupied.
5. In the last step, the operations of the restore-code are scheduled. The generated schedule is shown in Figure 6.15d.

6.6 Experiments and Evaluation

In this section, the performance of the proposed register assignment method is measured and evaluated. The target TTAs used in the experiments are instances of the TTA_{ideal} and the $\text{TTA}_{realistic}$ processors (see Section 4.2.2). The instances of each processor only differ in the number of integer registers. The largest model supports 512 registers. The smallest model still requires 10 registers. The reason is twofold: from the set of registers, seven registers are reserved by the compiler front-end as special registers (for the stack pointer, frame pointer and parameter passing). In addition to these seven registers, early assignment requires at least three registers for spilling[6]. Integrated assignment can always find a solution when the target TTA contains at least seven registers, because no reserved registers are needed for spilling. For the experiments a single RF for each register type (integer, floating-point and Boolean) is used. In Chapter 9, this restriction is released.

To measure the performance of the proposed integrated assignment

[6]These three extra registers are required to hold the memory addresses and the results of the reload operations in case all three operands of, for example, a multiply-add are spilled.

method the benchmarks are first compiled to sequential move code and simulated with representative data sets. The sequential code is scheduled for the target architecture with the use of profiling information. The last step combines the information from the parallel code with the profiling information in order to compute, for example, the cycle count. The performance numbers are obtained by averaging the speedups over all benchmarks.

Since both instruction scheduling and register assignment are NP-complete problems, heuristics are used for guiding the instruction scheduling and register assignment process. The heuristics are defined at three hierarchical levels:

- Register selection (Section 6.6.1)
- Operation selection (Section 6.6.2)
- Basic block selection (Section 6.6.3)

To evaluate the heuristics in a structured manner, they are evaluated independently. We are aware of the fact that the heuristics are not independent. However, evaluating all combinations of heuristics is too time-consuming. Furthermore, it is not to be expected that a set of heuristics exists, which in all situations, outperforms the others. In the final section, the performance of integrated assignment is compared with the best approach found in Chapter 5.

6.6.1 Register Selection

When an operation n that referes to a not already assigned variable v is scheduled, then a non-interfere register set is constructed for v. Each member of this set can be mapped onto a v. This set is defined as:

$$R_{\textit{Non-interfere}}(v, n) = R - R_{\textit{Interfere}}(v, n) \tag{6.16}$$

where R is the set of registers of the target processor of the required type (integer, floating-point, etc.). The set $R_{\textit{Interfere}}(v, n)$ is computed with equation 6.15.

Normally, this set has more than one element. The question arises which register to select. Integrated assignment has three choices: (1) mapping the variable onto a caller-saved register, (2) mapping the variable onto a callee-saved register or (3) spilling the variable to memory. The cost functions for each of these three choices are given in the Equations 3.5 (caller-saved cost), 3.6 (callee-saved cost) and 3.4 (spill cost). We propose to choose the category with the lowest cost. When the spill cost is the lowest, the variable is spilled to memory even when registers are available. When one of the other two categories has the lowest cost, a register is selected from the associated part of the register set. When no register in such a register set can be found, the category is chosen which has the second lowest costs. If this also fails no register can be found and the variable is spilled to memory.

6.6.2 Operation Selection

As discussed in Section 3.4.1, operations are selected for scheduling when they become a member of the $ready$ set. A heuristic is used to select an operation from this set. The basic block scheduler uses the $priority_{slack}$ heuristic as defined in Equation 3.10, which favors operations in the critical path. However, when registers are scarce it may be profitable to use a modified operation selection heuristic, which decrease the register pressure [GWC88]. This may reduce the amount of spill code and hence results in an improved performance. The goal of such a heuristic is to favor operations in the $ready$ set that will decrease the number of live ranges, i.e. select operations that end the live range of variables. This set of operations is denoted as $O_\perp \subseteq ready$. Three heuristics are proposed that increase the priorities of the operations in the set $O_\perp$. The priorities of all other operations ($ready - O_\perp$) remain unchanged.

- Step:

  $$priority_{step}\,(o \in O_\perp) = priority_{slack}\,(o) + \begin{cases} \delta & : \; |R_{free}| < \alpha \\ 0 & : \; |R_{free}| \geq \alpha \end{cases}$$

  where R_{free} is set of available registers in the last instruction of the currently scheduled basic block. This priority function increases the priority of operations in $O_\perp$ when $|R_{free}|$ drops below a certain threshold α.

- Linear:

  $$priority_{linear}(o \in O_\perp) = priority_{slack}(o) \cdot \begin{cases} 1 + \delta \left(\frac{\alpha - |R_{free}|}{\alpha} \right) : |R_{free}| < \alpha \\ 1 : |R_{free}| \geq \alpha \end{cases}$$

  where α is the register limit that determines when the priority should be increased. The parameter δ determines how strong the number of available registers $|R_{free}|$ influences the priority.

- Exponential:

  $$priority_{exponential}\,(o \in O_\perp) = priority_{slack}\,(o) \left(1 + \delta \cdot e^{-\frac{R_{free}}{\alpha}} \right)$$

  This priority scheme favors operations in $O_\perp$ using an exponential function. The parameters α and δ determine the impact of the number of available registers $|R_{free}|$ on the priority.

The impact on the priority of all three heuristics is shown in Figure 6.16. As can be clearly seen, the heuristics have more influence when the number of available registers decreases.

For all three heuristics a large number of experiments are performed while varying the parameter values. Unfortunately, the experiments showed that changing the priority function has little impact on the resulting performance.

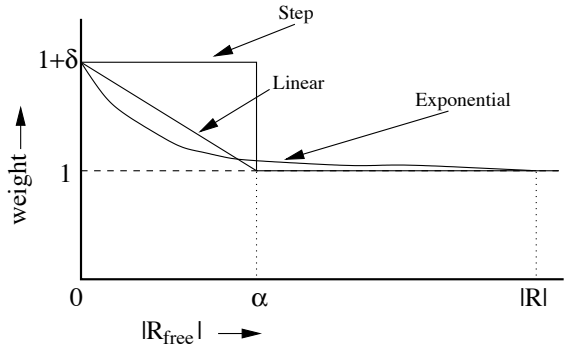


Figure 6.16: The priority as a function of the number of available registers for the three heuristics.

Only when registers are extremely scarce, a change in the priority scheme is profitable. However, the performance gain is very small (in the order of 0.1%). Changing the operation priority scheme, when a sufficient amount of registers is available may even hurt performance. Analysis of this unexpected behavior resulted in the following observations:

- The size of the $ready$ set is usually small. Since $O_\perp$ is a subset of $ready$, this set is even smaller. Consequently, only few operations are subject to a change in priority.
- A change in the priority when registers were scarce did not result in an increased performance, because the operations that were selected due to the new heuristic were selected anyway with the slack priority function. The same operations were selected independent of the priority function.
- Changing the priority hurts performance when operations that are not in the critical path are selected first. This was especially true when the priority was increased when sufficient registers (> 3) were available.

The above observations result in the conclusion that the slack priority heuristic, when using integrated assignment, not only favors critical operations but also does a fair good job in controlling the register pressure.

6.6.3 Basic Block Selection

In all experiments, the procedures of the benchmark programs are scheduled independently. Decisions made in one procedure do not influence decisions in other procedures. The next hierarchical level in a basic block scheduler is the basic block. When basic block scheduling and register assignment are done in separate phases, the order in which the basic blocks are scheduled has no influence on the resulting code. The basic blocks are scheduled as independent pieces of code. When, however, instruction scheduling and register assignment

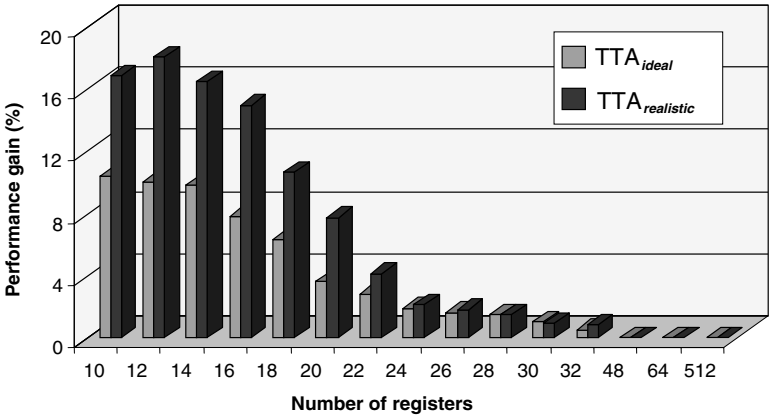


Figure 6.17: Speedup of the BB_{freq} heuristic compared to the BB_{top} heuristic.

are integrated into a single phase, the scheduling order of the basic blocks does matter. Register assignment decisions made during instruction scheduling in one basic block can influence scheduling/assignment decisions in other basic blocks since the live ranges of variables cross basic block boundaries. References in basic blocks that are scheduled early in the process can choose from all registers, while references in basic blocks that are scheduled later on only can pick registers from a smaller set of available registers. Register assignment for these later references is more likely to be hindered by a shortage on registers. The introduction of false dependences and spill code might be necessary.

In this section, two basic block priority functions are proposed and evaluated. Profiling information is used for determining the execution frequencies of the basic blocks. The evaluated heuristics are listed below.

BB_{top} Basic blocks are selected for scheduling in topological order; a basic block is selected when all its predecessor basic blocks are scheduled.

BB_{freq} The basic blocks are ordered according to their execution frequency. Basic blocks, which are executed more frequently, are scheduled first. Consequently, the set of free registers is larger in the most frequently executed basic blocks. This results in a large scheduling freedom, since registers do not hinder the construction of an efficient schedule in these code parts.

When multiple basic blocks conform to the used requirement the selection is done randomly. The assumptions concerning entry and exit basic blocks as discussed in Section 6.5.1 are respected.

Figure 6.17 compares the performance of both methods. The speedup numbers are obtained by averaging the speedups over all benchmarks. As can be seen from the results, the basic block scheduling heuristic has a large impact on the performance of integrated assignment. When registers are not a critical resource, the methods give approximately the same results. However, when reg-

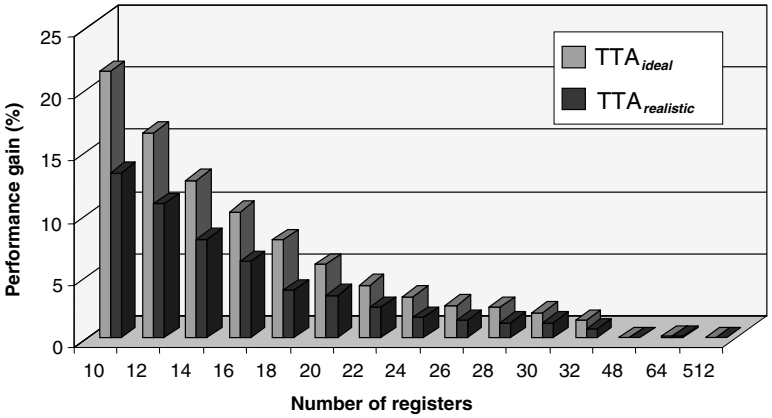


Figure 6.18: Speedup of integrated assignment compared to DCEA.

isters become scarce the execution frequency conscious heuristic outperforms the topological approach with more than 10%.

The results show a larger performance gain for the $\text{TTA}_{realistic}$ than for the TTA_{ideal}. This effect is caused by the fact that the amount of resources like FUs and buses are more limited for the $\text{TTA}_{realistic}$. When applying the BB_{top} heuristic with a limited set of registers, spill code is inserted in code with a high execution frequency such as loops. These extra operations require extra resources such as FUs. When these resources are limited, the operations cannot be executed in parallel, hence the size of the scheduled basic blocks is increased. When the resources are unlimited, this increases will be less or non-existing. When applying the BB_{freq} heuristic, spill code is inserted in code with a low execution frequency. Because the $\text{TTA}_{realistic}$ has a limited set of resources, these basic blocks also enlarge. However, this does not hurt performance as much as the BB_{top} heuristic does. For the TTA_{ideal} there are a large number of resources and hence the spill code can be scheduled in parallel when other dependences will permit this. Consequently, the performance penalty when applying the BB_{top} heuristic is less pronounced.

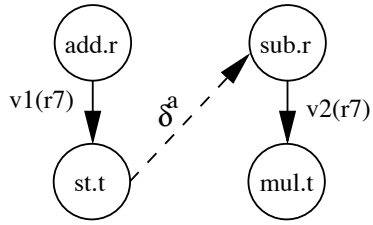
It is interesting to note that the performance gain for 10 registers is lower than for 12 registers for the $\text{TTA}_{realistic}$. This effect can be explained by the fact that when a large amount of spill code is needed, it will be placed in basic blocks with a high frequency count anyway.

6.6.4 Early vs. Integrated Assignment

To evaluate the introduced integrated assignment method, we compare its results, for each of the benchmarks described in Section 4.1, with the results of DCEA (Dependence-Conscious Early Assignment), the best register assignment method of Chapter 5. The averages of theses measurements for both target TTAs are shown in Figure 6.18 (the results of the individual benchmarks

```
add.r → v1;
v1 → st.t;
sub.r → v2;
v2 → mul.t;
```

a) Code fragment.



b) DDG with a false dependence.

```
add.r → st.t;
sub.r → mul.t;
```

c) Early assignment schedule

```
add.r → st.t;  sub.r → mul.t;
```

d) Integrated assignment schedule

Figure 6.19: False dependence effect when registers are scarce.

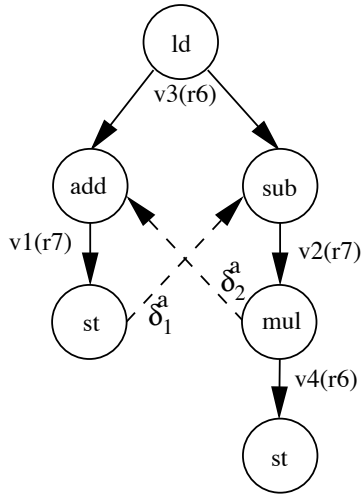
can be found in Appendix A). The number of registers in each target TTA is placed along the x-axis. The speedup of integrated assignment compared with DCEA is listed along the y-axis. The average performance gain varies between 0% for 512 registers and 21.3% for 10 registers.

As can be observed from the figure, the speedup is substantial for low register counts. As already demonstrated in Section 5.1, DCEA successfully tries to avoid false dependences; however, when registers become a critical resource it becomes difficult to avoid false dependences. This is illustrated in Figure 6.19. Figure 6.19a shows a small code fragment consisting of four transports. When sufficient registers are available, DCEA will prevent a false dependence and assigns different registers to `v1` and `v2`. However, when registers are scarce this is no longer guaranteed. To prevent spilling of other variables, or in an attempt to prevent other false dependences, DCEA may assign the same register to both variables `v1` and `v2`. This results in a false dependence in the DDG as depicted in Figure 6.19b. Although during scheduling, software bypassing and dead-result move elimination can, and often will be applied, the false dependence in the DDG prevents an optimal schedule. The resulting schedule is given in Figure 6.19c. The optimal schedule is given in Figure 6.19d. With integrated assignment it is possible to generate the optimal schedule even when registers are scarce, because it can re-use the registers freed due to software bypassing and dead-result move elimination.

The direction of a false dependence added by an early assignment approach, such as DCEA, is determined by the operation order in the sequential code. This is depicted in Figure 6.20. The sequential code is shown in Figure 6.20a. When the variables `v1` and `v2` are mapped onto the same register, a false dependence is added to the DDG. There are two possibilities, δ_1^a and δ_2^a, as shown in Figure 6.20b. Only one of them needs to be added. Adding false dependence δ_1^a results in a critical path of 5 instructions, while adding δ_2^a results

```
#100 → ld.t
ld.r → v3
v3 → add.o   #4 → add.t
add.r → v1;
v1 → st.o;   #100 → st.t
v3 → sub.o   #4 → sub.t
sub.r → v2;
v2 → mul.o   #7 → mul.t
mul.r → v4;
v4 → st.o;   #104 → st.t
```

a) Code fragment.



b) DDG with two possible false dependences.

Figure 6.20: Direction of false dependences.

in 4 instructions in the critical path, assuming single cycle latencies. Apparently, false dependence δ_2^a is preferable. Unfortunately, early assignment will add δ_1^a due to the operation order in the sequential code. A solution could be to take this observation into consideration when adding false dependences prior to scheduling. However, during scheduling the critical path may change due to scheduling decisions, and thus the effect of the selected false dependence may turn out not to be advantageous. In other words, the order of operations and the introduced false dependences hinders out-of-order scheduling. Because integrated assignment assigns registers during scheduling, it allows reordering of the operations and can adapt to new situations caused by scheduling decisions.

Another effect, which emerges when registers become scarce, is the interaction between register assignment and instruction scheduling. The scheduler selects operations for scheduling according to a priority function, which favors the operations in critical paths. Because integrated assignment assigns a register to a variable when the first operation that refers to this variable is scheduled, it respects the operation ordering of the instruction scheduler. In other words, for operations in the critical path it is more likely to find a free register. Early assignment does not interact with the instruction scheduler and may introduce false dependences and spill code in the critical path.

The results in Figure 6.18 show that the performance gains for the TTA_{ideal} are larger than the performance gains for the $\text{TTA}_{realistic}$. When applying early assignment for the $\text{TTA}_{realistic}$, false dependences can be concealed by a shortage on resources such as FUs and buses. In other words, some operations are

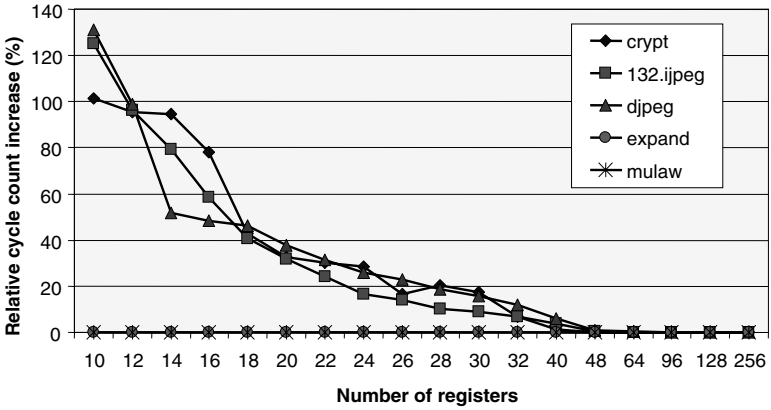


Figure 6.21: Cycle count increase relative to the TTA_{ideal} with 512 registers while applying DCEA.

scheduled sequentially anyway due to a resource conflict independent of the presence of a false dependence. This is not the case for the TTA_{ideal}; a false dependence is not hidden by a resource shortage since a large number of resources are provided. Because a false dependence introduced by early assignment when scheduling for the TTA_{ideal} is more visible than when scheduling for the $\text{TTA}_{realistic}$, the performance gain for the TTA_{ideal} is larger.

As can be seen in the tables in Appendix A, the performance gains of the benchmarks differ significantly. The benchmark *crypt*, *132.ijpeg* and *djpeg* have large performance gains while the benchmarks *expand* and *mulaw* do not show any improvement. Figure 6.21 shows the cycle count increase of DCEA relative to the TTA_{ideal} with 512 registers for the five mentioned benchmarks. The relative cycle count for the benchmarks *crypt*, *132.ijpeg* and *djpeg* increases when the number of registers decreases. These benchmarks require a large number of registers, and thus opportunities are present to allow integrated assignment to improve performance. The relative cycle count for the benchmarks *expand* and *mulaw* is not influenced when the number of registers is reduced. These benchmarks only require 10 registers and thus integrated assignment cannot improve their performance.

In some situations, also a small negative performance effect can be observed as shown in the tables in Appendix A. This is mainly caused by the register selection heuristics. Both methods, DCEA and integrated assignment, use heuristics to determine whether to assign a caller-saved or callee-saved register, or to spill the variable. In some situations, this choice leads to a negative performance impact. The effect is larger for the $\text{TTA}_{realistic}$ than for the TTA_{ideal} because the inserted spill code requires extra resources. When resources are limited, the impact is larger. However, on average, integrated assignment outperforms DCEA.

6.7 Conclusions

Based on the observations of Chapter 5, an integrated assignment method is developed in combination with a basic block scheduler. In this chapter, the general principle of this innovative method is presented. A method is developed, which constructs a set containing all registers that can be mapped onto a variable v, while part of the code is already scheduled, and part of the variables are already assigned to registers. This set contains instruction precise information about available registers. The introduced method does not add false dependences prior to scheduling. During scheduling/assignment, false dependences are only created when there are not enough registers. In contrast to early assignment methods, the number of required registers is reduced by exploiting software bypassing in combination with dead-result move elimination. In order to be complete, specific issues related to the insertion of spill code and the insertion of caller-saved and callee-saved code are addressed. Furthermore, several heuristics are presented and evaluated.

The method is compared with the best early assignment approach found in Chapter 5. All methods are implemented within the same compiler, which makes a fair comparison possible. The experiments showed performance gains up to 100%. Especially when registers were scarce, integrated assignments outperformed DCEA. Since we compared our method with the best approach implemented in Chapter 5, we conclude that integrated assignment also outperforms late assignment and strictly early assignment.

Although we discussed the new method in the context of TTAs, we believe that it is also applicable for superscalars and VLIWs. A problem is, however, that these architectures do not suppress unnecessary register write-backs and thus spill code cannot be scheduled without the use of registers. To overcome this problem, one can reserve a small set of registers for spilling. Alternatively, an extension for superscalar processors, as proposed in [LG95], could alleviate this problem. Lozano and Gao describe a hardware scheme that avoids the commits of variables that are only live in the reorder buffer. This is similar to software bypassing, with the advantage that dependent instructions do not have to be scheduled in the same cycle in order to avoid commits, which relaxes the scheduling process and may result in even larger speedups.

Integrated Assignment and Global Scheduling 7

The amount of exploitable ILP in basic blocks is limited. To justify the duplication cost of FUs and data paths in ILP processors, the ILP between operations of different basic blocks should also be exploited. In general, more exploitable ILP increases the number of simultaneously live variables in the schedule produced. Consequently, the register pressure increases and registers should be assigned with more care. In this chapter, an extension of integrated assignment is proposed, which fully integrates register assignment and region scheduling.

This chapter is organized as follows. In Section 7.1, the construction of the interference register set is discussed when operations are imported into basic blocks. The algorithm for importing operations is presented in Section 7.2. Section 7.3 presents an example of the proposed method. When integrated assignment runs out of registers, a decision has to be made whether to insert spill code or to schedule the code less aggressively. This is discussed in Section 7.4. The insertion of state preserving code in a region scheduler and its consequences are described in Section 7.5. In Section 7.6, various register selection heuristics are presented to increase the performance of the generated code. The last section evaluates the proposed techniques and states the conclusions.

7.1 The Interference Register Set

An extended basic block scheduler increases the exploitable ILP by moving operations over basic block boundaries. For reasons discussed in Section 3.4.4, importing operations may result in code duplication. Importing and duplica-

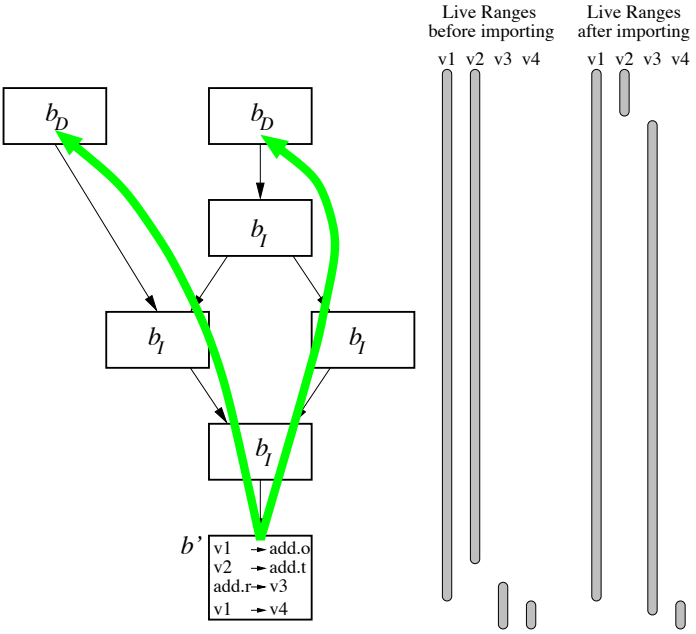


Figure 7.1: Stretching and shrinking of live ranges when importing all the moves of the addition.

tion changes the live ranges of the variables referenced by the imported operation. This is shown in Figure 7.1. Importing the addition of basic block b' to the basic blocks b_D, results in a shorter live range for the variable `v2`, while the live range of `v3` is stretched. The live range of `v1` does not change, because it is required by the copy operation in the source basic block b'.

Importing operations changes the number of basic blocks spanned by the referenced live ranges. When a live range is stretched, the register allocator must check additional basic blocks for a legal assignment. The opposite holds for live ranges that shrink. The consequences of shrunk and stretched live ranges on the computation of the interference register set are discussed in respectively Section 7.1.1 and Section 7.1.2.

7.1.1 Importing a Use

Importing an operation n, which uses a variable v, may result in a shorter live range for v. For a legal register assignment, the live-variable information must be updated. The imported operation n is removed from basic block b', therefore the sets $live_{Use}(b')$ and $live_{Def}(b')$ are recomputed. Operation n is added to the duplication basic blocks $b_D \in D$. This has the following consequence for the $live_{Use}$ sets of these basic blocks.

$$live_{Use}(b_D) = \begin{cases} live_{Use}(b_D) \cup \{v\} & : \quad v \notin live_{Def}(b_D) \\ live_{Use}(b_D) & : \quad otherwise \end{cases} \quad \forall \; b_D \in D \qquad (7.1)$$

The basic blocks on all paths from the duplication basic blocks ($b_D \in D$) to the source basic block (b') are called *intermediate basic blocks* ($b_I \in I$).

$$I = \{b \in B \mid b_D \rightsquigarrow b \wedge b \rightsquigarrow b', b_D \in D\} \qquad (7.2)$$

where $b \rightsquigarrow b'$ means that there is a control flow path within the region from b to b'. The live-variable information in these intermediate basic blocks may change also when operation n is imported. It is tempting to simply remove the variable v from the sets $live_{In}$ and $live_{Out}$ of all basic blocks $b_I \in I$. Unfortunately, sometimes the intermediate basic blocks have references to v, or v is an element of the set $live_{In}$ of one of the successors (excluding b' and the intermediate basic blocks) of the intermediate basic blocks. In these situations, the live range shrinks only partly. The new live-variable information in the intermediate basic blocks can be computed with the Equations 3.1 and 3.2. Note, only the live-variable information of the basic blocks $b \in \{b'\} \cup I \cup D$ changes. Only for these basic blocks, the equations have to be solved.

Additional steps are required when variable v is already mapped onto a register r. The RRVs that are not spanned anymore by the new shorter live range, incorrectly indicate that r is already used and cannot be assigned to another variable. This hinders an efficient register assignment. The RRVs of the intermediate basic blocks $b_I \in I$, the source basic block b' and the duplication basic blocks $b_D \in D$ must be updated. Register r is removed from all RRVs of a basic block $b \in \{b'\} \cup I$ if $v \notin live_{In}(b) \wedge v \notin live_{Def}(b)$. The RRV information in the not yet scheduled duplication basic blocks remains unchanged because before and after importing, v is live in these basic blocks. The RRV information in the already scheduled duplication basic blocks is updated in the same way as if the use was scheduled with local scheduling.

The new live-variable and RRV information is computed prior to importing. This information reflects the situation as if operation n referring to v is imported in the duplication basic blocks b_D[1]. Consequently, the same method for constructing the interference register set as in basic block scheduling can be used (see Equation 6.15).

7.1.2 Importing a Definition

Importing an operation n, defining a variable v, stretches the live range of v. The basic blocks spanned by the new live range consist of the basic blocks of the original live range, the intermediate basic blocks and the duplication basic blocks. Importing operation n changes the live-variable information. To

[1]When the live range does not change, neither the live-variable nor the RRV information needs to be updated.

produce a legal register assignment, the sets $live_{Use}(b')$ and $live_{Def}(b')$ are recomputed. The set $live_{Def}$ of the duplication basic blocks changes also:

$$live_{Def}(b_D) = \begin{cases} live_{Def}(b_D) \cup \{v\} & : \; v \notin live_{Use}(b_D) \\ live_{Def}(b_D) & : \; otherwise \end{cases} \quad \forall \; b_D \in D \tag{7.3}$$

In addition, the $live_{In}$ and $live_{Out}$ sets of the basic blocks $b \in \{b'\} \cup I \cup D$ change. These sets can be computed with Equations 3.1 and 3.2. However, there is a simpler method to compute these new live sets. For the source basic block b' only the set $live_{In}$ changes:

$$live_{In}(b') = live_{In} \cup \{v\} \tag{7.4}$$

The duplication basic blocks b_D require only an update of the set $live_{Out}$:

$$live_{Out}(b_D) = live_{Out}(b_D) \cup \{v\} \quad \forall \, b_D \in D \tag{7.5}$$

Because the intermediate basic blocks $b_I \in I$ do not contain any references to v, otherwise importing operation n was illegal[2], their live information can simply be computed with:

$$live_{In}(b_I) \;=\; live_{In}(b_I) \cup \{v\} \quad \forall \, b_I \in I \tag{7.6}$$

$$live_{Out}(b_I) \;=\; live_{Out}(b_I) \cup \{v\} \quad \forall \, b_I \in I \tag{7.7}$$

The new live-variable information is computed prior to importing. It reflects the situation as if the reference to v is imported in the duplication basic blocks.

The construction of the interference register set as given in Equation 6.15 assumes that all references to variable v are located in not yet scheduled basic blocks. When operations are imported, they can also be added to already scheduled duplication basic blocks[3]. Let's denote this set of scheduled duplication basic blocks $D^+ \subseteq D$. The ordering of the operations in these basic blocks is completely known. This allows us to make a more accurate register availability estimation. The interference register set can now be computed with an equation similar to Equation 6.11, where i_{cur} is replaced with the earliest instruction $EarliestInsn(b, n)$ in which the duplicated definition can be scheduled. Dependence constraints are used for computing the earliest instruction.

$$R^+_{RRV}(b, n) = \bigcup_{i=EarliestInsn(b,n)}^{LastInsn(b)} RRV(i) \tag{7.8}$$

[2]Operation n would not be selected for importing, because of a false dependence between the operation referring to v in one of the intermediate basic blocks and operation n.

[3]This is caused by the fact that guarded expressions, in the current implementation, are only computed for already scheduled duplication basic blocks. When, for example, a variable is off-live, importing may fail because one of the duplication basic blocks is not yet scheduled. When all duplication basic blocks are scheduled or are being scheduled, the guarded expressions can be computed and importing may succeed. As a result, the operation is imported into already scheduled basic blocks.

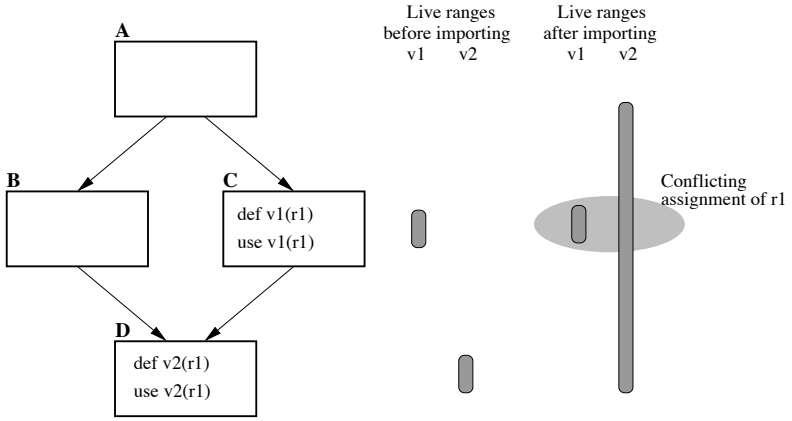


Figure 7.2: Register conflict when operations are imported.

The interference register set for a variable v defined by an imported operation n can now be computed with:

$$R_{Interfere}(v,n) = \left(\bigcup_{b \in B_{IO}(v)} R_{IO}(v,b) \right) \cup \left(\bigcup_{b \in B_{DU}(v) - D^+} R_{DU}(v,b) \right) \cup \left(\bigcup_{b \in D^+} R^+_{RRV}(b,n) \right) \tag{7.9}$$

Note that due to importing $I \subseteq B_{IO}$ and $D \subseteq B_{DU}$.

A refinement can be made when computing the interference register sets for the basic blocks $b \in D^+$. It can happen that the register requirements of the earliest instruction where the definition can be scheduled, in combination with the register requirements of the other parts of the live range, exceed the number of available registers. When the definition is scheduled in a later instruction, the register pressure may reduce. Therefore, the definition is annotated with the earliest instruction in which the register requirements are still met. The scheduler uses this information to determine the earliest instruction in which a scheduling attempt is made.

The previous discussion assumes that variable v is not yet mapped onto a register. This is, however, not always true. When a register was already assigned before importing, the situation can arise that the assigned register is not free in the stretched live range. This is illustrated in Figure 7.2. Assume that the definition of `r1` in basic block **D** is imported into basic block **A**. As a consequence, in basic block **C**, two variables are live simultaneously that are assigned to the same register. This will result in incorrect program execution. Consequently, importing is illegal. However, it may turn out that another register can be used

Algorithm 7.1 TRYTOIMPORTOPERATION(b, o)

```
b' = BB(o)
D = COMPUTEDUPLICATIONSET(b, b')
UPDATEUSEINFORMATION(b', D, o)
UNASSIGNDEFINITION(o)
UPDATEDEFINFORMATION(b', D, o)
FOR EACH b'' ∈ D DO
  IF b'' ∈ is_scheduled THEN
    IF ¬ TRYTOSCHEDULEOPERATION(b'',o) THEN
      RELEASERESOURCES(D, o)
      RESTOREDEFINFORMATION(b', D, o)
      REASSIGNDEFINITION(o)
      RESTOREUSEINFORMATION(b', D, o)
      return
    ENDIF
  ENDIF
  b'' = b'' ∪ {o}
ENDFOR
b' = b' − {o}
```

for the new stretched live range. Assigning this register to variable v2 allows the code motion. Therefore, prior to computing the interference register set, the assignment of the definition is undone, and the same procedure is followed as if the variable was not assigned. When it turns out that not sufficient resources are available to successfully import the operation, importing fails and the original assignment is redone.

7.2 Importing Operations

The implementation of integrated assignment into the region scheduler requires some changes to Algorithm 3.8 as shown in Algorithm 7.1. Prior to importing, the live-variable and RRV information is changed, as if the operation was imported, using the functions UPDATEUSEINFORMATION and UPDATEDEFINFORMATION. As discussed in the previous section, an earlier assignment of the definition may hinder the algorithm to find a free register. Therefore, the assignment of the definition is undone with the function UNASSIGNDEFINITION. When all information is updated, the scheduler attempts to import operation o in the duplication basic blocks D. After an operation has been imported successfully, the RRV information is updated. When however, due to some resource constraint this is not feasible, all scheduling decisions made so far for this operation, and the changed live-variable and RRV infor-

mation, must be restored to their original state.

When a variable has definitions or uses outside the scheduled region, the RRVs in the live-range that are outside the region are also updated. This is in contrast with the approach chosen by [FR91], where special data structures are needed to distribute the register assignment information from one trace to another. In our approach, the register assignment information is implicitly distributed to other regions.

7.3 Example

An extensive example is used to demonstrate the operation of the proposed method. We start this example with the CFG given in Figure 7.3. It shows the schedule before the addition in basic block **E** is imported into the duplication basic blocks **A** and **B**. It is assumed that basic block **A** is currently being scheduled and basic block **B** is already scheduled. Due to earlier scheduling steps, both basic block **A** and **B** contain three instructions (for reasons of clarity, the already scheduled operations are not shown). The basic blocks are annotated with live-variable and RRV information. Variable `v1` is already mapped onto register `r3`. This assignment is reflected in the RRVs of all shown basic blocks. Because the basic blocks **A** and **B** are already scheduled, and `v1` is live on en-

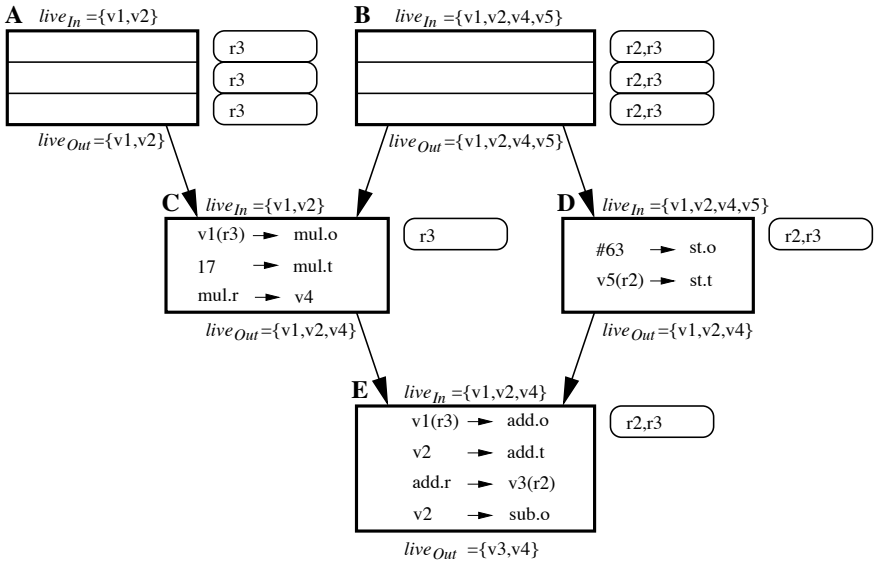


Figure 7.3: CFG prior to importing.

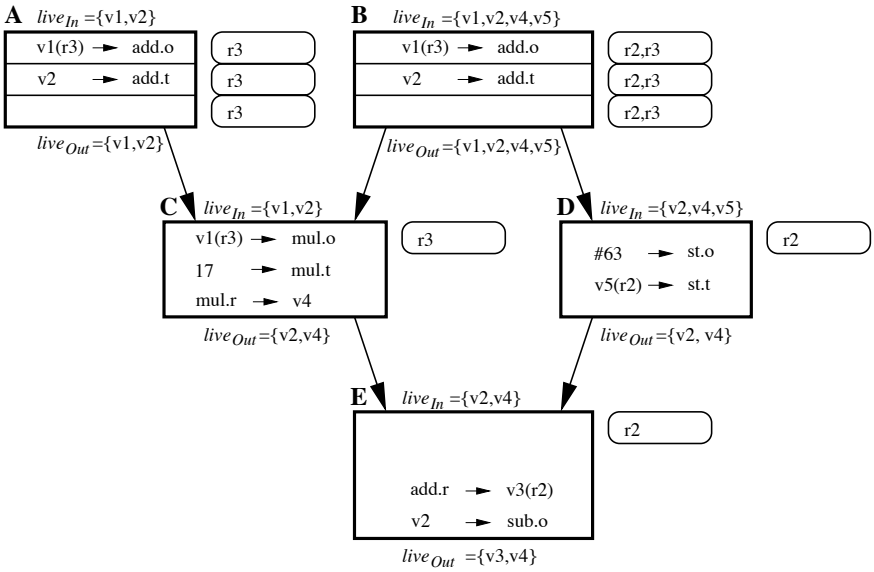


Figure 7.4: CFG as if the operand and trigger move of the addition were imported.

try of both basic blocks, all their RRVs contain register `r3`. The other basic blocks (**C**, **D** and **E**) are not scheduled yet. They have only one RRV with register `r3` set as unavailable.

Importing the addition results in a shorter live range for variable `v1`. Because basic block **C** contains a reference to `v1`, the live range cannot shrink completely. The live range of variable `v2` does not shrink at all, because basic block **E** contains another use of `v2`. Figure 7.4 shows the situation as if the moves `v1(r3)` → `add.o` and `v2` → `add.t` were imported. In addition to the changes in the live-variable information, some RRVs are also changed. Because `v1` is not live anymore in the basic blocks **D** and **E**, register `r3` is removed from their RRVs.

Importing the transport `add.r` → `v3(r2)` stretches the live range of variable `v3`. Note that `v3` was already assigned to register `r2`. This register, however, is also used in basic block **D** by variable `v5`, which makes importing illegal. To allow importing, the assignment of `v3` is undone. Figure 7.5 shows the result of unassigning `r2`, and the new live-variable and RRV information, as if the move `add.r` → `v3` is imported.

In the last step, the addition and its duplicate are scheduled. First, the trigger and operand moves are scheduled. During scheduling, register `r1` is assigned to variable `v2`. This register is added to the RRVs spanned by `v2`'s new

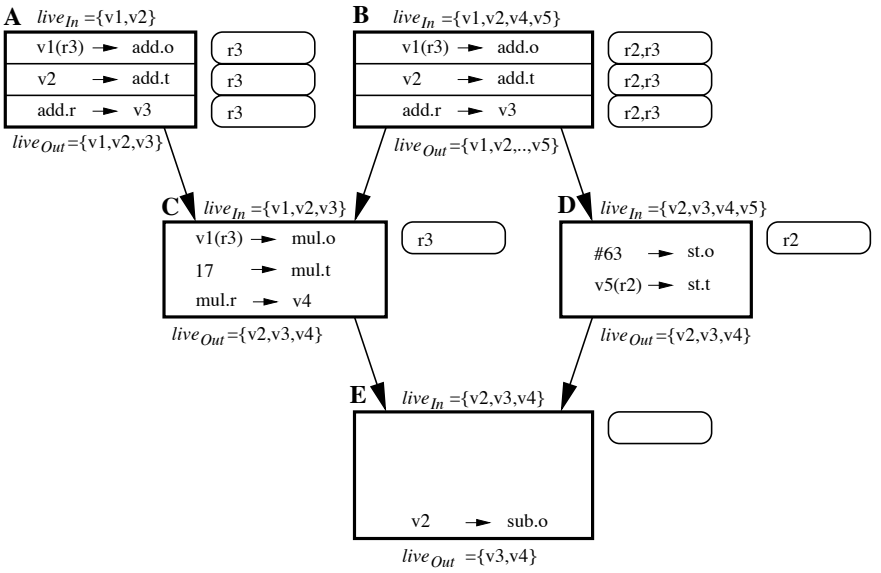


Figure 7.5: CFG as if the result move of the addition is imported.

live range. Because of this assignment, and the previously recorded register assignment information in the RRVs, the interference register set for `v3` becomes $\{$`r1`, `r2`, `r3`$\}$. In the example, `r4` is selected and assigned to variable `v3`. The resulting scheduled code, including the updated information in the RRVs, is shown in Figure 7.6.

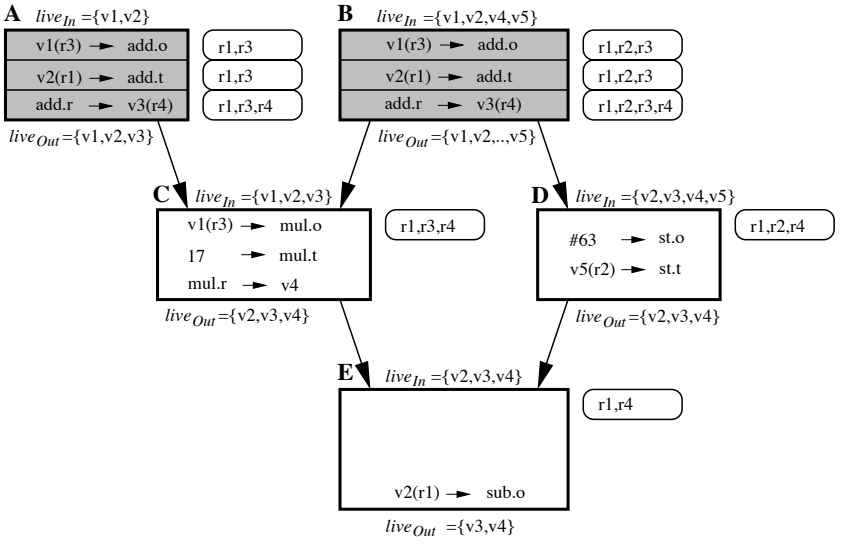


Figure 7.6: CFG after importing.

7.4 Spilling

Importing fails when during importing it is discovered that insufficient resources, including registers, are available for the to be scheduled operation and its duplicates. When a shortage of registers is the cause for failing, the insertion of spill code can solve this problem. Both type of references, uses and definitions, have their own specifics in relation to spilling.

The decision whether to spill a variable that is *defined* by a to be imported operation, is based on the following observations:

- The live range of a definition (result move) is lengthened when the operation is imported, and hence the number of interferences increases. When a register can be found for the original live range but not for the stretched live range, it is unclear whether spilling will be advantageous.
- Generating spill code for each definition for which no register can be found, results in a similar problem as with late assignment: importing is applied too aggressively. This results in a large amount of spill code. The inserted spill code also needs extra resources, which leads to inefficient code.
- When other operations are imported, registers might become available and another attempt can be made.

Based on the above observations, it was decided not to insert spill code when no register can be found for an imported definition. Instead, importing is rejected. However, when another operation is successfully imported, another attempt is made to import the operation since the register pressure might be reduced.

Importing an operation may result in shorter live ranges for the variables *used* by it. In general, a shorter live range has less interference with other live ranges. Therefore, the probability to find a register for this shorter live range is larger than for the original longer live range. In our approach, spill code is inserted when no register can be found for the new live range. The motivation to insert spill code instead of rejecting importing is the following: (1) because no register could be found for the shorter live range, it is very unlikely that a register can be found for the original live range, and thus the variable is spilled anyhow, (2) it might be possible that all inserted spill operations, including the operation itself, can be imported because the register shortage problem was located elsewhere in the code. Therefore, when importing fails due to insufficient registers for the uses, the necessary reload code is inserted in the source basic block b' just before the operation. In subsequent scheduling steps, it is tried to import the individual operations of the reload code sequence. This can be problematic, because extra registers are required for the new live ranges that are local to the reload code sequence. In Section 6.4.3, it has been shown that the problem of extra registers for scheduling reload code sequences can be

overcome by scheduling the complete reload code sequence in a single step. Unfortunately, importing a complete reload code sequence is complex and has a high probability of failure, because it is very unlikely that the complete reload code sequence fits into the available instructions of all duplication basic blocks. Our choice, importing the operations of the reload code sequences individually (and thus assigning registers to the live ranges introduced by spilling), has as a consequence that incomplete reload code sequences may remain in source basic block b'. Therefore, the algorithm for scheduling reload code sequences is modified in such a way that it can also schedule the incomplete reload code sequences that are left in the source basic blocks.

In general, importing spill code and reload code is handled in the same way as importing other operations. The incomplete spill code and reload code sequences that are left in the source basic blocks are scheduled as if they were a single operation, i.e. software bypassing and dead-result move elimination are enforced when no registers are available.

7.5 State Preserving Code

The insertion of callee-saved code is normally applied when all variables are mapped onto registers. Only at that point it is known, which registers need to be saved. This is problematic for an integrated assignment approach. In Section 6.5.1, a method has been described in the context of basic block scheduling, which inserts callee-saved code in the exit and entry basic blocks of a procedure, when all other basic blocks have been scheduled. A similar approach is used for region scheduling: callee-saved code is inserted in the entry and exit basic blocks when the outermost region is selected for scheduling. Because regions are scheduled from inner to outer, all other regions are already scheduled, and variables referenced in these regions are already mapped onto registers. Similar to the approach presented in Section 6.5.1, an early assignment step is applied to map variables in the outermost region onto callee-saved registers. Afterwards, the callee-saved code is inserted. This approach allows importing of the callee-saved code inserted in the exit basic blocks, which might improve performance.

Procedure calls are never subject to importing, because it is unlikely to find empty instructions for the call delay slots in all already scheduled duplication basic blocks. Scheduled duplication basic blocks only contain empty instructions when the delay of an operation is longer than a single cycle. However, inserting a procedure call between the trigger and the result of an operation results in incorrect program behavior. Inserting extra empty instructions is also not an option for the same reasons as discussed in Section 5.2.1.

Since procedure calls are never imported, it seems logical to use the same approach for scheduling procedure calls as in basic block scheduling. Unfortunately, this leads to less parallel code than possible. This is shown in Fig-

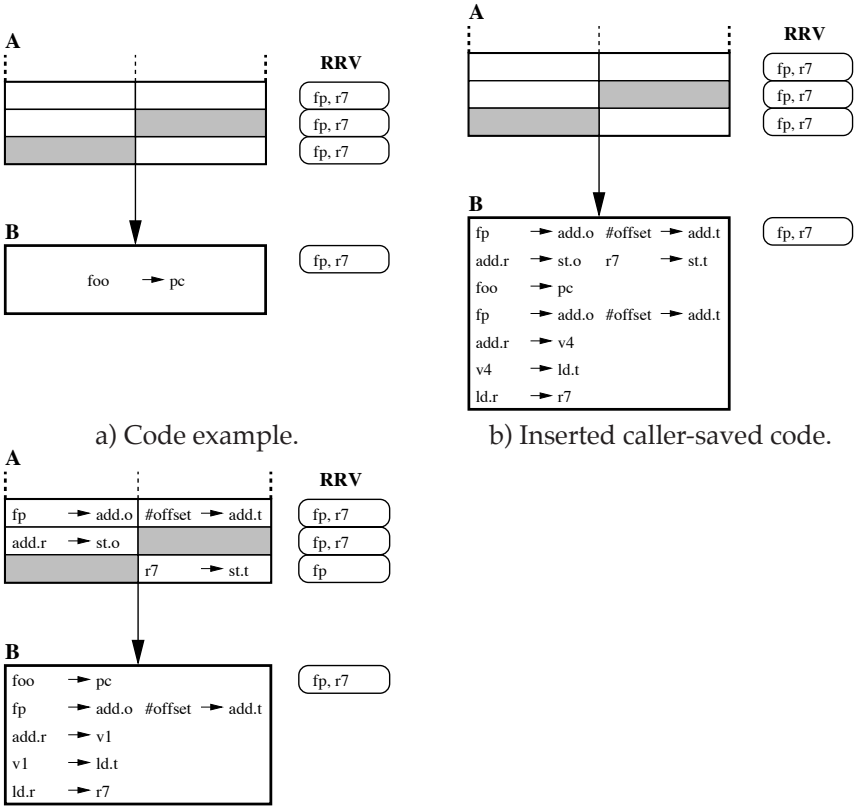


Figure 7.7: Importing procedure calls.

ure 7.7. Figure 7.7a shows a code fragment in which it is assumed that basic block **A** is already scheduled, and all operations that could be imported are imported. The next basic block to be scheduled is **B**. Selecting the procedure call for scheduling results in the generation of caller-saved code in the same way as described in Section 6.5.2. The generated caller-saved code is now scheduled in basic block **B**. This is shown in Figure 7.7b. However, the generated operations could have been candidates for importing, which could increase performance. The problem is that the caller-saved code is generated too late.

Selecting procedure calls for importing without actually importing them solves this problem. When the region scheduler selects a procedure call for importing, caller-saved code is generated in the basic block of the procedure call. This code is not yet scheduled and subject to importing. Consequently, the generated caller-saved code can be imported. This is shown in Figure 7.7c.

Due to importing of other operations, other live ranges can become live across not yet scheduled procedure calls. This does not create extra problems since

each time an operation is successfully imported, all other operations for which importing failed, including procedure calls, are tried again. When a procedure call is selected again for scheduling, caller-saved code is generated for these new live ranges too. In other words, caller-saved code is generated incrementally.

7.6 Experiments and Evaluation

In this section, the proposed integrated assignment method is evaluated in combination with region scheduling. In order to optimize the performance, various heuristics are proposed and evaluated. In Section 7.6.1 the selection order of regions is addressed. So far, the decision to spill a variable v was based on the ratio of the spill cost, caller-saved cost and callee-saved cost. In Section 7.6.2 a new heuristic is proposed, which also takes the spill cost of the variables interfering with v into account. In the last section, the performance of integrated assignment is compared with DCEA, the best register assignment approach found in Chapter 5.

7.6.1 Region Selection

Region scheduling traditionally starts with scheduling the inner most regions first. When assigning registers for these inner regions, all registers are available and spilling is less likely to occur. However, the inner most regions are not always the most heavily executed regions. Another approach is to schedule the most frequently executed region first. Consequently, spill code is pushed to the less frequently executed regions. This should potentially result in a better performance. In this section, two region selection heuristics are compared:

1. $\mathsf{R}_{\mathsf{top}}$: A region is selected for scheduling when all its inner regions are scheduled.
2. $\mathsf{R}_{\mathsf{freq}}$: The most frequently executed regions are scheduled first.

Note that within a region, the basic blocks are always scheduled in topological order.

The results of our experiments indicated that the $\mathsf{R}_{\mathsf{freq}}$ heuristic only slightly outperforms the $\mathsf{R}_{\mathsf{top}}$ heuristic, on average 0.5%. This seems surprisingly low, because a similar heuristic for local scheduling resulted in much higher performance gains (Section 6.6.3). Fortunately, this difference can easily be explained. Many of the inner regions are also the most frequently executed regions. Therefore, the cycle count of many benchmarks is equal, independent of the region selection method.

7.6.2 Global Spill Cost Heuristic

The approach used so far to decide whether to spill variable v, only takes the costs associated with v into account. It ignores the spill costs of the variables that are simultaneously live with v. A variable v, referenced by an operation that is scheduled at an early stage in the scheduling process, has a high probability of being mapped onto a register. Variables referenced by operations that are scheduled in a later stage often are spilled to memory. However, it may turn out that the spill cost associated with these variables is higher than the spill cost of v, and hence spilling them may have a negative impact on performance.

The above discussion results in the conclusion that the spill costs of variables that are simultaneously live with v should also be taken into account. Therefore, a *global spill cost heuristic* (GSC) is proposed. At each point (instruction) in the live range of v, the set of interfering variables is computed. When the number of not yet assigned variables with higher spill cost than v exceeds the number of available registers at a point in v's live range, it may be advantageous to spill v. Unfortunately, the begin- and end-points of many live ranges are unknown because the references to the variables, including v, are not yet scheduled. Therefore, it is impossible to accurately compute the set of interfering variables at each point in the live range of v. To tackle this problem, the definition of a point is changed from instruction to basic block. Only the basic blocks in which v is live on entry and exit are taken into consideration. The set of not yet assigned variables with a higher spill cost than v in a basic block b, can now be computed with:

$$live_{Spill}(v,b) = \{v_i \in live_{bb}(b) - v \mid C_{spill}(v_i) > C_{spill}(v) \wedge r(v) = \emptyset\} \quad (7.10)$$

where $live_{bb}(b) = live_{In}(b) \cup live_{Def}(b)$ is the set of all variables live within basic block b. The function $r(v)$ returns register r that is mapped onto variable v. This function returns the empty set when no register is assigned to v. The set of available registers in a basic block b is computed with:

$$R_{Avail}(b) = R - \bigcup_{i=0}^{LastInsn(b)} RRV(i) \quad (7.11)$$

The global spill cost heuristic can now be described as: a variable v is spilled to memory when $\mid live_{Spill}(v,b) \mid > \mid R_{Avail}(b) \mid$ for any basic block b in the live range of v.

During the experiments, it was also noticed that importing operations works too greedy. As a result, too many live-ranges were stretched and no more registers were available in the intermediate basic blocks I. This leads to excessive spill code in these basic blocks. Fortunately, the GSC heuristic also controls the register pressure in these basic blocks. An operation defining a variable v is *not* imported when the GSC heuristic evaluates to true in any of

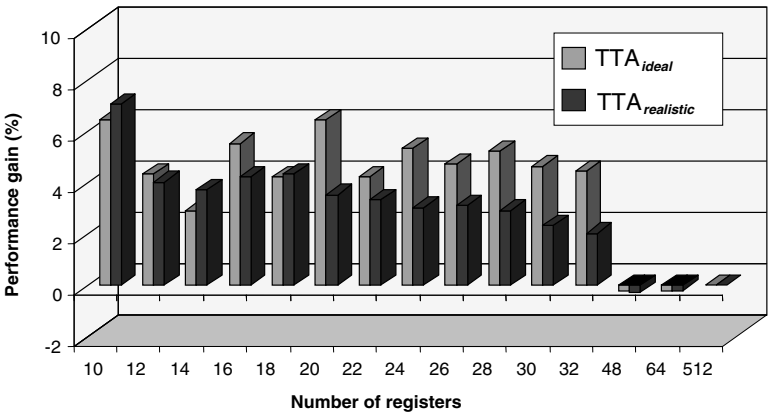


Figure 7.8: Performance impact of the GSC heuristic.

the basic blocks spanned by the new (stretched) live range. It is of course not necessary to apply above heuristic, when due to importing the number of live ranges in the intermediate basic blocks remains constant or decreases. Operations that do not define a variable are never restricted for importing, because importing these operations does not increase the register pressure. The performance impact of applying the GSC heuristic is shown in Figure 7.8. Applying this heuristic is indeed beneficial.

One could argue that the GSC heuristic is too conservative. It ignores the fact that scheduling rearranges the code and doing that also changes the interference relations. Furthermore, the set $live_{Spill}(v, b)$ is too conservative. This set contains all variables referenced within a basic block b. It is, however, very unlikely that all these variables are live simultaneously and thus the register pressure is probably lower. In addition, software bypassing and dead-result move elimination may remove variables from the $live_{Spill}(v, b)$ set. In order to take these effects into account, it seems a good idea to relax the GSC heuristic. Therefore, we propose only to use the GSC heuristic when the number of available registers $R_{Avail}(b)$ drops below a certain threshold n. The new heuristics are now denoted as $\mathsf{GSC_n}$. Our experiments indicated that $n = 5$ resulted, on average, in the highest performance. The performance gains when applying the $\mathsf{GSC_5}$ heuristic compared to the GSC heuristic are shown in Figure 7.9.

7.6.3 Early vs. Integrated Assignment

The key question is: how does the introduced method compare in terms of cycle counts to the best register allocator of Chapter 5? Figure 7.10 shows this performance comparison when all benchmarks are scheduled using a region scheduler. The TTA_{ideal} and $\mathrm{TTA}_{realistic}$ templates are used with a varying number of registers. As can be seen, integrated assignment outperforms

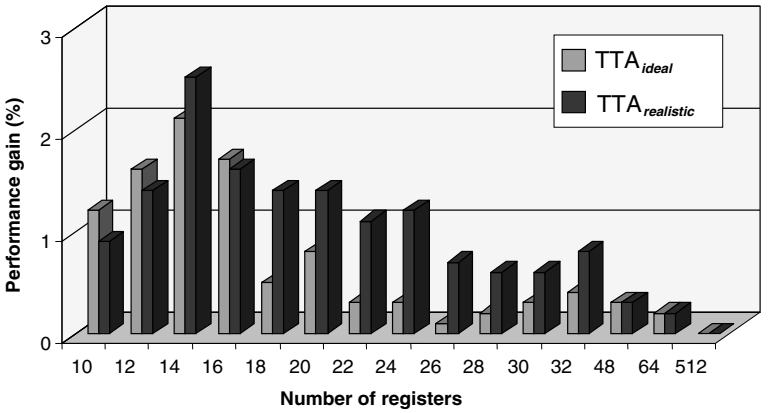


Figure 7.9: Performance impact of the GSC_5 heuristic.

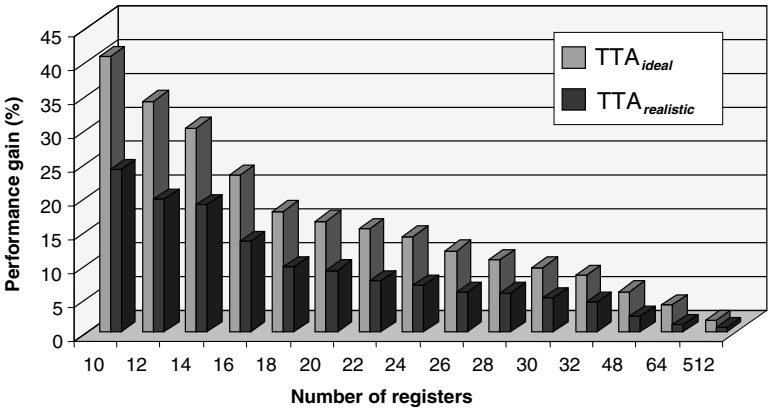
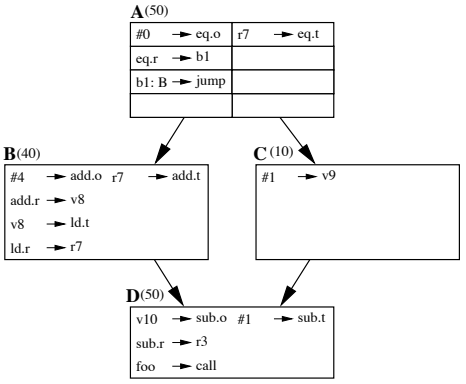


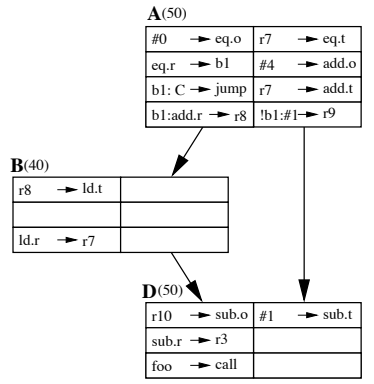
Figure 7.10: Speedup of integrated assignment compared to DCEA using region scheduling and the TTA_{ideal} and $TTA_{realistic}$ templates.

DCEA. Especially when registers are scarce, the performance improvement is large. The results of the individual benchmarks can be found in Appendix A.

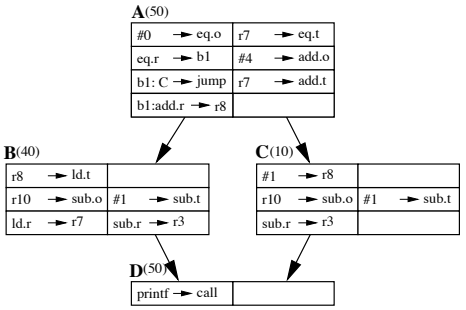
The TTA_{ideal} template is used to show the performance increase of the method without being hindered by shortage of resources other than registers. Consequently, a large amount of ILP could be exploited. This resulted in a high register pressure. A method, such as integrated assignment, that efficiently exploits the available registers gives better results when the register pressure is high. In practice, the TTA_{ideal} template will never be used because of it cost and high cycle time. However, it shows the potential of the method. A more practical TTA template is the $TTA_{realistic}$. The resources of this template are more restricted and hence less ILP can be exploited. As shown in Figure 7.10, the performance gain is still substantial. However, it is lower than when using the TTA_{ideal} template.



a) CFG.



b) CFG after applying integrated assignment.



c) CFG after applying DCEA.

Figure 7.11: Too aggressively false dependence avoidance.

As already observed in Section 6.6.4, various reasons exist to explain the performance differences between applying DCEA and integrated assignment for local scheduling. The same reasons can be applied to region scheduling. However, the performance gains are larger. Because region scheduling exploits a larger amount of ILP, the register pressure increases. A method that uses the available registers more efficiently can achieve a higher performance.

In addition to the reasons mentioned in Section 6.6.4, another effect is observed. DCEA only considers forward false dependences, see Section 5.1.3. This is a valid assumption, since this restriction avoids the addition of many edges into the interference graph, which almost never will hinder efficient code generation. However, sometimes these neglected potential false dependences are important and hinder the generation of efficient code. This is the case for the benchmarks *132.ijpeg* and *mpeg2encode* where DCEA shows a small performance loss even when sufficient registers (512) are available. This effect can also work the other way around. Consider the CFG of Figure 7.11a. All operations in basic block **A** are already scheduled and the scheduler is ready to import operations into basic block **A**. Figure 7.11b shows the completely sched-

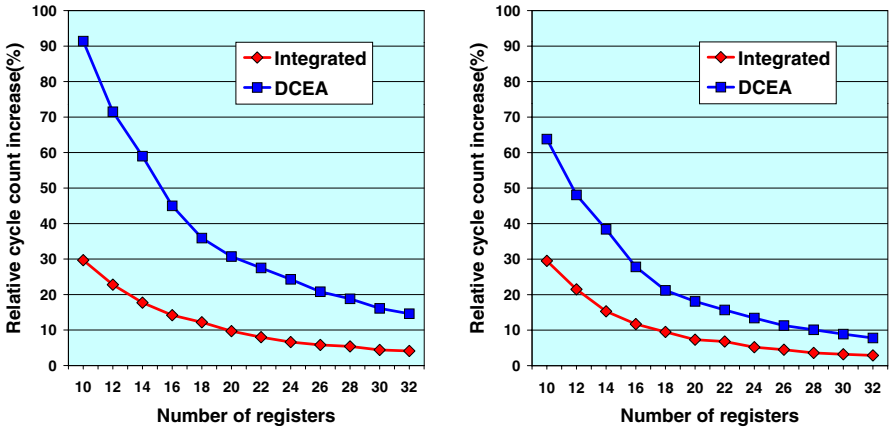
uled CFG when integrated assignment is used. This method was so successful in importing operations that no operations remain in basic block **C**. As a result, this basic block is removed from the CFG. This seems advantageous, however, this is not always true. When basic block **C** is removed from the CFG, it is impossible to import operations from basic block **D** to basic block **B**, although sufficient resources are available in basic block **B**. During code examination, we discovered that DCEA sometimes performed better in these situations. Since DCEA only considers forward false dependences, it may map variable `v8` and `v9` onto the same register, for example `r8`. This results in a false dependence between the addition of basic block **B** and the copy of basic block **C**. Because of this dependence, it is no longer possible to import the copy operation and hence basic block **C** is not removed from the CFG. This has the advantage that it is now possible to import the subtraction of basic block **D** into basic block **B** and its duplication basic block **C**. The resulting scheduled CFG is shown in Figure 7.11c. Assuming the execution frequencies as given within parentheses, the CFG will execute in 470 cycles when applying integrated assignment and in 400 cycles when using DCEA. Summarizing, although integrated assignment is good in avoiding false dependences, it may result in a performance loss. The negative performance of the benchmark *expand* is caused by this effect. Future solutions are to postpone the deletion of empty basic blocks or to (re)insert empty basic blocks.

The reduction of required callee saved code also contributes to the small performance gain when sufficient registers are available. Because integrated assignment uses the available registers more efficiently, it requires less callee-saved code, which results in a small performance gain.

As discussed in the previous section, the GSC_5 heuristic is used to limit the aggressiveness. In some situations, this leads to a negative performance gain, because the aggressiveness is limited too much. As an example consider the *wc* benchmark. When registers are scarce, a negative performance gain resulted for the TTA_{ideal} template. When no heuristic is used to limit the aggressiveness, a speedup can be observed (see Appendix A). However, the use of the GSC_5 heuristic resulted, on average, in the best results.

We were also curious as to how severe reducing registers hurts performance. This data is shown in Figure 7.12; it shows the cycle count increase relative to a configuration with 512 registers; all benchmarks are averaged. For TTAs with 32 registers, integrated assignment already outperforms DCEA. When the number of registers is reduced, the performance gap grows to more than 60%.

TTAs were originally proposed for application specific purposes; therefore it is also interesting to know how many registers can be saved while the performance stays constant. Fewer registers, results in a smaller chip area and in faster access to the register file, both are important for application specific processors. In Table 7.1, a comparison is made between integrated assignment and DCEA, to find out what these savings are. We see that integrated assign-



a) TTA_{ideal}. b) $TTA_{realistic}$.

Figure 7.12: Relative cycle count increase of integrated assignment and DCEA.

ment needs substantially fewer registers than DCEA and, therefore, is the best solution for embedded systems.

Besides the number of registers required for a certain performance, also the code size of the application is an important factor for embedded systems. Memories on these systems are usually limited. Performance optimizations that increase the code size to a large extent, are not suitable for these systems. Figure 7.13 shows the impact on the code size when using integrated assignment compared to DCEA. This figure clearly demonstrates that integrated assignment reduces the code size. This comes not as a surprise, because instruction scheduling can be done more efficient and thus the number of instructions required to pack the transports reduces.

Table 7.1: Register requirements of integrated assignment and DCEA with equal performance.

TTA_{ideal}			$TTA_{realistic}$		
Integrated assignment	**DCEA**	**increase[%]**	**Integrated assignment**	**DCEA**	**increase[%]**
10	20	100 %	10	16	60 %
12	25	108 %	12	18	50 %
14	29	107 %	14	22	57 %
16	32	100 %	16	26	63 %
24	48	100 %	18	29	61 %
24	> 64	-	20	32	60 %
10			28	48	71 %
20			32	> 48	- %

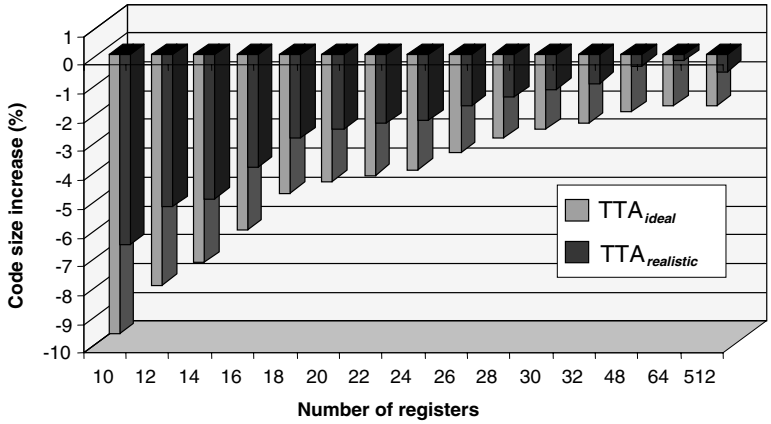


Figure 7.13: Increase in code size of integrated assignment compared to DCEA using the TTA$_{ideal}$ and TTA$_{realistic}$ templates.

7.7 Conclusions

In this chapter, integrated assignment as presented in Chapter 6 is extended to a larger scheduling scope. The impact of importing operations is discussed. The principle of importing operations is used in many scheduling scopes, such as regions, traces, superblocks, hyperblocks and decision trees. Although the discussion focuses on a region scheduler, the discussed techniques can also be applied in combination with alternative scheduling scopes.

Besides the impact of stretching and shrinking of the live ranges of variables referenced by imported operations, also spilling and the insertion of state preserving code has been presented. Heuristics were proposed and evaluated. Limiting the import greediness is one of the major concerns when applying integrated assignment. When importing is applied aggressively, the register pressure may increase, which on its turn may result in a large amount of spill code and thus hurts performance. On the other hand, when importing is restricted to a great extent, the amount of exploited ILP decreases, which also hurts performance. The use of the global spill cost heuristic helps to control the import greediness. It can be tuned to achieve the best performance.

Integrated assignment in combination with a region scheduler outperforms all other evaluated late and early assignment methods. It needs substantially fewer registers than other methods to achieve the same performance. This results in a saving of silicon area. Furthermore, the code size decreases in comparison with DCEA.

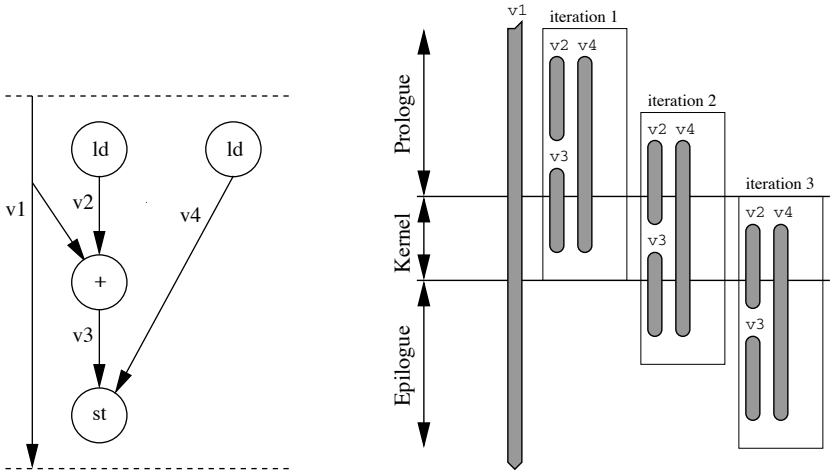
Integrated Assignment and Software Pipelining

8

Software pipelining is a powerful and efficient scheduling technique to exploit instruction level parallelism (ILP) in loops. In this chapter, integrated assignment in the context of software pipelining is discussed. Software pipelining schedules loops such that each iteration in the produced schedule consists of operations chosen from different iterations in the sequential code. The generation of an optimal resource-constrained schedule for loops is known to be NP-complete [Lam88, GJ79]. Heuristics are used for guiding the construction of the schedules in polynomial time.

The disadvantage of aggressive scheduling techniques, such as software pipelining, is their resulting high register pressure. To obtain a valid schedule the compiler must be able to match the register requirements with the available registers of the target processor. When more registers are needed, some additional actions are required, such as the insertion of spill code.

In this chapter, a new technique is proposed, which integrates register assignment and iterative modulo scheduling. In Section 8.1, the relation between register pressure and software pipelining is described. In Section 8.2 related approaches are discussed that perform register assignment in a separate step. The proposed technique is presented in Section 8.3. Section 8.4 gives the experimental results and evaluates the method. The conclusions are stated in Section 8.5.



a) DDG of loop-body. b) Live ranges in software pipelined loop.

Figure 8.1: Live ranges of variables in a software pipelined loop.

8.1 Register Pressure

The register pressure of a software pipelined loop is equal to the number of simultaneously live variables in the loop. The problem of assigning registers for software pipelined loops is somewhat different from other code, because variables, which were live in one iteration in the original loop, can be live in multiple iterations in the software pipelined version. This section explores some characteristics of the live ranges of variables in software pipelined loops. Two types of variables can be distinguished:

- *Loop-invariant* variables are live in the loop, but are never modified during loop execution.
- *Loop-variants* variables are modified in each iteration of the loop. The modified value can be used by an operation in the same or other iteration.

The loop in Figure 8.1a has three loop-variants `v2`, `v3`, and `v4`, and one loop-invariant `v1`. When an isolated iteration of the original loop is considered, live ranges `v2` and `v3` never interfere and can be mapped onto the same register. As a result, three registers are required to hold the variables: two registers to hold the loop-variant variables and one for the loop-invariant variable `v1`. In software pipelined loops, live ranges that did not interfere in the original loop interfere in the scheduled code. This is illustrated in Figure 8.1b. The live range of variable `v2` overlaps with the live range of variable `v3` of the previous iteration. To obtain a valid schedule, they have to be mapped onto distinct registers. The register pressure is thus increased.

Another aspect of software pipelining is the presence of *long-living-variables*. The live ranges of these variables are longer than the initiation interval II. Un-

less special measures are taken, this results in illegal schedules since new values are generated before previous ones are used. The variable v4 in Figure 8.1b is such a long-living-variable, it overlaps with itself in a previous iteration. To deal with this problem some form of renaming must be applied so that successive definitions of v4 use distinct registers. This renaming can be performed at compile time or in hardware.

One approach to apply renaming at compile time is *modulo variable expansion* [Lam98]. This technique schedules the loop prior to register assignment. During scheduling, the inter-iteration false dependences between references to the same variable are ignored. The resulting incorrect schedule is unrolled to ensure that each live range fits in the II. After unrolling, variable renaming is applied to correct the schedule. The minimum degree of unrolling (K_{min}) is determined by the longest live range of all loop-variants in the kernel and the II. K_{min} can be calculated as

$$K_{min} = \max_{\forall v} \left\lceil \frac{L(v)}{II} \right\rceil \tag{8.1}$$

where $L(v)$ represents the length of the live range of v.

A *rotating register file* [BYA93, RYYT89] is a hardware solution to deal with the problem of long-living-variables. Dedicated hardware solves the problem without replicating code. It renames different instantiations of a loop-variant at execution time. A rotating register file rotates the registers every loop iteration. Register r_i becomes r_{i+1} and r_n becomes r_0. This allows a value, generated by an operation in one iteration, to co-exist with the values generated by the same operation in previous and subsequent iterations.

Both modulo variable expansion and rotating register files have their drawbacks. Modulo variable expansion increases the static code size and implies late assignment. As discussed in Chapter 5, late assignment is not the best choice for compiling programs for TTAs, therefore this method is rejected. Rotating register files require dedicated hardware. This may affect the cycle time and requires extra hardware. In our opinion, the extra hardware costs are better spent on more general-purpose registers, because then the complete program can benefit.

In [Hoo96], another technique is proposed to deal with long-living-variables: *delay lines*. Delay lines are the software counterpart of rotating register files. They are implemented as a series of copy operations. The delay lines are inserted before instruction scheduling and register assignment. The number of delay lines per variable is determined by $\left\lfloor \frac{L(v)}{MII} \right\rfloor$. Because the code is not yet scheduled, the precise length of the live ranges is unknown. The length of a live range is defined as the longest path in the DDG from the definition to the consumer of the variable. Figure 8.2 shows the effects of the insertion of a delay line. The live range of variable v4 is split in two (v4 and v5). As a result, five registers are required to hold all variables of the software pipelined loop. We use delay lines to solve the problem of long-living-variables.

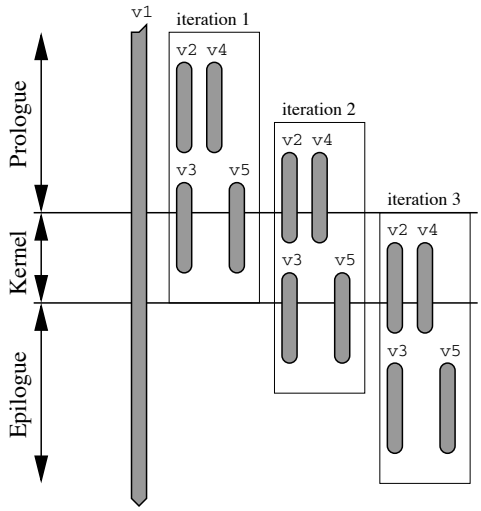


Figure 8.2: Live ranges when using delay lines.

All of the three mentioned methods for handling long-living-variables lead to an increased register pressure. When more registers are required than available, the register allocator fails to find a solution. To obtain a schedule, which respects the register constraints, additional actions are required. In the remainder of this section, methods to decrease the register pressure in software pipelined loops are discussed.

Consider the loop in Figure 8.3a. For reasons of clarity, RISC style code is used. This loop executes in five cycles per iteration assuming an issue width of three, a latency of two for the multiply and a single cycle latency for the other operations. Three registers are needed when it is assumed that the variables `v1` and `v3` are loop carried. The software pipelined version of this loop, including its register assignment, is given in Figure 8.3b. This loop executes in two cycles per iteration, however, the register pressure is increased to four. The result registers of the load and the multiply interfere in the software pipelined version, while this was not the case in the original loop. When the II is increased with one, again only three registers are needed. The generated schedule with an $II = 3$ is given in Figure 8.3c. Reasons why the register pressure decreases when the II increases are:

- Fewer live ranges interfere since less iterations of the original loop overlap.
- The number of long-living-variables decreases. As a consequence, the number of delay lines decreases and thus fewer registers are required.

The principle of trading performance against a lower register pressure is generally applicable for software pipelining, however, this process cannot be extended indefinitely. The loop in the previous example always requires at least

```
L1: add  v1, v1, #4
    ld   v2, (v1)
    mul  v4, v2, #17
    sub  v3, v3, #4
    st   v4, (v3)
    bgz  v3, L1
```

a) Loop body.

```
Prologue
    add r1, r1, #4
    ld  r2, (r1)
    add r1, r1, #4   mul r4, r2, #17
    ld  r2, (r1)     sub r3, r3, #4
Kernel
L1: add r1, r1, #4   mul r4, r2, #17  st  r4, (r3)
    ld  r2, (r1)     sub r3, r3, #4   bgz r3, L1
Epilogue
                     mul r4, r2, #17  st  r4, (r3)
                     sub r3, r3, #4
                                      st  r4, (r3)
```

b) Software pipeline with $II = 2$, $R_{press} = 4$.

```
Prologue
    add r1, r1, #4
    ld  r2, (r1)
    mul r2, r2, #17
Kernel
L1: add r1, r1, #4   sub r3, r3, #4
    ld  r2, (r1)                    st,  r2(r3)
    mul r2, r2, #17                 bgz r3, L1
Epilogue
                     sub r3, r3, #4
                                    st  r2(r3)
```

c) Software pipeline with $II = 3$, $R_{press} = 3$

Figure 8.3: Increasing the II to reduce register pressure.

three registers. Reasons why increasing the II does not reduce the register pressure indefinitely are:

- The live range of loop-invariants in the loop is always II instructions. Independently of the schedule, they require one register each.
- Each loop-body has a minimal register requirement. Note that this register requirement can drop to zero for TTAs when software bypassing and dead-result move elimination are applied.

The second alternative to reduce the register pressure is spilling. However, the addition of spill code may increase the II, which reduces performance as well. Adding spill code in software pipelined loops is discussed in detail in Section 8.3.2. The last method to reduce the register pressure is to split the loop into two or more loops. Smaller loops tend to use fewer registers. Loop splitting or other loop restructuring techniques in relation to software pipelining are outside the scope of this thesis and are not investigated any further.

```
L1: #4     →  add.o   v1; →  add.t
    add.r  →  v1
    v1     →  ld.t
    ld.r   →  v2
    #17    →  mul.o   v2; →  mul.t
    mul.r  →  v4
    #4     →  sub.o   v3; →  sub.t
    sub.r  →  v3
    v3     →  st.o    v4; →  st.t
    #0     →  gt.o    v3; →  gt.t
    gt.r   →  b0
    b0: jump L1
```

a) TTA code of Figure 8.3a.

```
                                        Prologue
    #4 → add.o;   r1 → add.t
    add.r → r1;   add.r → ld.t
    #4 → add.o;   r1 → add.t;   ld.r → mul.t;   #17 → mul.o
    add.r → r1;   add.r → ld.t; #4 → sub.o;     r3 → sub.t
                                        Kernel
L1: #4 → add.o;   r1 → add.t;   ld.r → mul.t;   #17 → mul.o;   mul.r → st.t;   sub.r → r3;
    sub.r → st.o; sub.r → gt.t; #0 → gt.o;      b0:jump L1
    add.r → r1;   add.r → ld.t; #4 → sub.o;     r3 → sub.t;     gt.r → b0
                                        Epilogue
                              ld.r → mul.t;   #17 → mul.o;   mul.r → st.t;   sub.r → st.o
                              #4 → sub.o;     r3 → sub.t
                                                             mul.r → st.t;   sub.r → st.o
```

b) Software pipelined TTA code.

Figure 8.4: Software pipelined TTA code with $II = 2$, $R_{press} = 2$.

8.2 Register Assignment and Software Pipelining

When register assignment is applied *prior* to software pipelining, the constraints added by register assignment prevent an efficient schedule. In Figure 8.4a, the loop of Figure 8.3a is translated into TTA code. It requires only three registers to hold all variables, because `v2` and `v4` can be mapped onto the same register. This assignment, however, results in a false dependence between the result moves of the load and the multiply. When local or global scheduling was used, this false dependence does not hinder the generation of an efficient schedule. However, when the loop is software pipelined, it increases the length of the kernel to three instructions in order to avoid interference between `v2` and `v4`.

In Section 5.1, techniques to prevent the creation of false dependences are presented. The TTA compiler back-end uses the forward parallel interference graph (IG_{fpar}), see Section 5.1.3. Applying this method straightforwardly for software pipelining does indeed prevent the creation of false dependences. However, only the *intra-iteration* false dependences are considered. The reg-

ister assignment of Figure 8.4a, respects all intra-iteration false dependences, it is, however, the inter-iteration false dependence between the result move of the load and multiply, which results in an II of three. To identify potential inter-iteration false dependences, the to be software pipelined loops are pre-software pipelined before register assignment, without considering resource constraints [Hoo96]. The resulting schedule is analyzed to see which code motions were carried out. The resulting interferences are added to the forward parallel interference graph in the same manner as described in Section 5.1.3. When all added interference edges are respected during early register assignment, the scheduler has more freedom to generate the optimal throughput schedule.

The example in Figure 8.4 requires four registers in the sequential code to avoid all false dependences. Figure 8.4b shows the resulting maximal throughput schedule. Careful examination of the code learns that only two registers are actually used in the parallel code. Although four registers are required to generate this schedule, the number of actual required registers has dropped to two, due to the ability of TTAs to software bypass values.

Spilling in the context of early assignment in combination with software pipelining is not a severe problem. Spill code generated by the register allocator is scheduled in the same way as the other operations, but of course may increase the II.

In contrast to early assignment, late assignment can exploit the software bypass property of TTAs. This reduces the number of required registers. Since scheduling is done before register assignment, the false dependences created by the register allocator do not hinder the generation of efficient code. Most published work on software pipelining assume late assignment. Applying software pipelining first can lead, however, to loops with a high register pressure. Various heuristics are proposed to decrease the register pressure.

Slack Scheduling [Huf93] schedules some operations late and other operations early, with the aim to reduce the register requirements and achieving maximum execution rate. The algorithm uses a heuristic, which integrates recurrence constraints and critical-path considerations in order to decide when each operation is scheduled.

In [ED95b], a method is proposed that tries to reduce the register pressure when the complete loop is scheduled. After scheduling, some operations are moved to an earlier or later instruction without changing the throughput of the schedule. Moving operations in a smart fashion can reduce the register requirements.

Hypernode Reduction Modulo Scheduling (*HRMS*) [LVAG95] is a heuristic strategy that tries to shorten loop-variant live ranges, without sacrificing performance. The main part of HRMS is the ordering strategy, which orders the nodes before scheduling them. This prevents that direct predecessors and successors, of a node are scheduled before the node itself is scheduled. Thus, only predecessors *or* successors of a node can be scheduled before the node itself is

scheduled, but not predecessors *and* successors. During scheduling, the nodes are scheduled as soon/late as possible, if predecessors/successors have been scheduled previously. The main drawback of HRMS is that it does not take into account that nodes are more critical in the scheduling process if they belong to a more critical path.

Swing Modulo Scheduling [LGAV96] considers latencies to decide how critical the nodes are. It gives the highest priority to operations in the most critical path. To reduce the register requirements each node is placed as close as possible to its predecessors and successors.

Applying software pipelining with the intention to achieve high throughput, without considering register requirements, may lead to impractical solutions. Most modulo scheduling approaches use heuristics to produce near-optimal schedules with reduced register requirements. All mentioned methods have in common that it is not guaranteed that the number of required registers is less than or equal to the number of available registers. None of the discussed methods inserts spill code. Inserting spill code in an already scheduled software pipelined loop, without affecting the whole schedule, is very difficult and in many situations impossible. The option used in the Cydra 5 compiler [DT93] is to start over again with an increased II. If after several attempts this also fails, conventional techniques, such as basic block scheduling, are used for scheduling the loop.

In [LVA96] an approach is presented, which does insert spill code. It is generated in an iterative way. First, a software pipelined schedule is generated, which tries to minimize the register pressure, then the assignment is done using graph coloring [Cha82, BCKT89]. When no valid assignment is found, spill code is inserted in the original code and the complete loop is rescheduled. Rescheduling is necessary, since the added load/store operations might not fit in the schedule.

Since the register requirements of a software pipelined loop are highly dependent on the way the loop is generated, it seems beneficial to apply register assignment and instruction scheduling of software pipelined loops in a single phase. Methods based on integer linear programming are proposed [NG93] to generate optimal schedules with register constraints. However, the number of computations is very large.

8.3 Integrated Assignment and Modulo Scheduling

In Chapter 6 and 7, we investigated integrated register assignment and instruction scheduling for basic block scheduling and region scheduling. This section describes how this method can be used in combination with software pipelining. In Section 8.3.1, the construction of the interference register set is discussed. Section 8.3.2 describes the instruments we have to decrease the register pressure, when the number of registers is too small to hold all variables.

8.3.1 The Interference Register Set

Integrated assignments in combination with software pipelining builds upon iterative modulo scheduling as has been presented in Section 3.4.5. The set of used registers per instruction is recorded using Registers Resource Vectors (RRVs). Just as for all other resources, register conflicts not only can arise in the same instruction i, but also in all instructions $i + II \cdot k \ \forall k > 0$. When an operation in a software pipelined loop is being scheduled, registers are assigned to the variables to which it refers. Checking whether a register is free for variable v, implies checking the RRVs of all instructions that are spanned by the live range of v. The part of the live range outside the software pipelined loop is checked in the same way as for local scheduling (see Section 6.3). In addition, *all* instructions, thus II instructions, of the software pipelined loop are checked. This seems very conservative. However, this is needed because in a software pipelined loop there is no way to tell where the other definitions or uses will be scheduled. Normally, the definition will be scheduled before the use, but with software pipelining this is not guaranteed. Therefore, the interference register set of the currently being scheduled software pipelined loop cannot be computed as accurate as being done with local and region scheduling. The interference register set for a variable v, which is first referenced in a software pipelined loop, is constructed with:

$$R_{Interfere}(v) = \left(\bigcup_{b \in B_{IO}(v) \cup B_{Swp}} R_{IO}(v,b) \right) \cup \left(\bigcup_{b \in B_{DU}(v) - B_{Swp}} R_{DU}(v,b) \right) \quad (8.2)$$

where B_{Swp} is the set of basic blocks that belong to software pipeline-able loops, including the currently scheduled basic block. Because the complete live range of variable v is included, no extra analysis or repair code is needed to integrate the software pipelined loop in the total schedule. Note, that the inclusion of the RRVs of all instructions in the II limits the re-usage of registers. This is not a problem in early and late assignment, however, these approaches have their own problems as discussed previously. Only when software bypassing and dead-result move elimination are applied, integrated assignment can re-use a register.

When the interference register set is constructed, the register allocator picks a register of the set of non-interfering registers $R_{\textit{Non-interfere}}(v) = R - R_{\textit{Interfere}}(v)$, assigns the selected register to the variable and updates all associated RRVs. If an operation is scheduled, which has already assigned variables, then to keep the bookkeeping precise, the RRVs of the loop are updated. In the context of software pipelining, this is only possible when all uses and definitions of the variable in the loop are scheduled, since only then it is precisely known in which instruction the variable is live.

Algorithm 8.1 SOFTWAREPIPELINING(b)

```
II = COMPUTEMII(b)
budget = budget_ratio · |b|
n = 0
WHILE ¬ ITERATIVEMODULOSCHEDULING(b, II, budget, Huff) ∧
        ¬ ITERATIVEMODULOSCHEDULING(b, II, budget, Rau) DO
  IF n > Threshold ∧ FAILEDDUEREGISTERCONSTRAINT(b) THEN
    var = SELECTVARIABLETOSPILL(b)
    GENERATESPILLCODE(var)
    n = 0
    II = RECOMPUTEII(b)
    budget = budget_ratio · |b|
  ELSE
    II = II + 1
    n = n + 1
  ENDIF
ENDWHILE
```

8.3.2 Spilling

The same method as described in Section 6.4.2 can be applied when inserting spill code. However, the DDG of a software pipelined loop has one peculiarity compared to the DDG of other code: inter-iteration dependences. If applicable, the inter-iteration dependence between the definition and the use of the spilled variable is replaced by an inter-iteration dependence between the store and the load operations. To prevent a value to be read from memory before it is written, the associated delay is set to one instruction. Just as discussed in Section 6.4.2, a second data dependence edge between the load and the store operation is added to prevent that a value is written to memory before it is read. In the same way as in local scheduling, the complete spill code and reload code sequences are scheduled in a single scheduling step by using software bypassing and dead-result move elimination. This has the advantage that no extra registers are required for the short live ranges created by spilling.

Besides spilling, there is another alternative to decrease the register pressure: increasing the II (see Section 8.1). This may result in a better schedule. Note that the introduction of spill code has as a side effect that the II may increase as well. These combined effects may result in an even larger decrease of the register requirements. In [LVA96], it is even claimed that the introduction of spill code is preferable above increasing the II. In order to investigate this claim, we propose a combination of adding spill code and increasing the II.

The proposed algorithm (see Algorithm 8.1) first performs integrated register assignment and iterative modulo scheduling without spill code. When this fails, the II is increased. If the II is increased with more instructions than a pre-

defined threshold and still no valid schedule is found, the algorithm considers spilling. If an operation could not be scheduled due to a register shortage, all scheduling and assignment decisions are made void, as if no operation in the loop is already scheduled. Next, the algorithm selects a live range for spilling using the heuristic of Equation 3.4. All live ranges are considered, including those who in the previous attempt received a register. The insertion of load and store operations changes the resource requirements and the precedence constraints. Trying to obtain a valid schedule within the original II, which is probably lower than the new one, is futile. Therefore a new II is computed and the process is repeated until a solution is found.

8.4 Experiments and Evaluation

In this section, integrated assignment in combination with software pipelining is evaluated. Not all loops are suitable for software pipelining. As discussed in Section 3.4.5, software pipelining can only be performed on some of the inner most loops. The remaining parts of the code are scheduled by the region scheduler.

In Section 8.4.1, the impact of changing the $Threshold$ (see Algorithm 8.1) is discussed. In Section 8.4.2, the integrated assignment approach is compared with DCEA.

8.4.1 Spilling or Increasing the II

As discussed in Section 8.3.2 there are two options to reduce the register pressure: (1) the insertion of spill code and (2) increasing the II. Algorithm 8.1 switches between increasing the II and the insertion of spill code. When the $Threshold$ is set to zero, the algorithm inserts spill code at the moment it discovers that insufficient registers are available to generate a schedule in II instructions. When $Threshold$ is for example set to five, the algorithm will first try to reduce the register pressure by increasing the II. When this fails after five attempts, the algorithm decides to insert spill code. In the experiments, we varied the $Threshold$ between 0 and 10. When sufficient registers were available, all experiments gave of course the same results. When registers were scarce, the $Threshold$ only influenced the performance of a few benchmarks. On average, no significant performance improvement was found. One reason for this behavior is that the inserted spill code often can be scheduled without increasing the II. This leads to the conclusion that most benchmarks are insensitive for a change in the value of $Threshold$ and that spilling is the best way to reduce the register pressure.

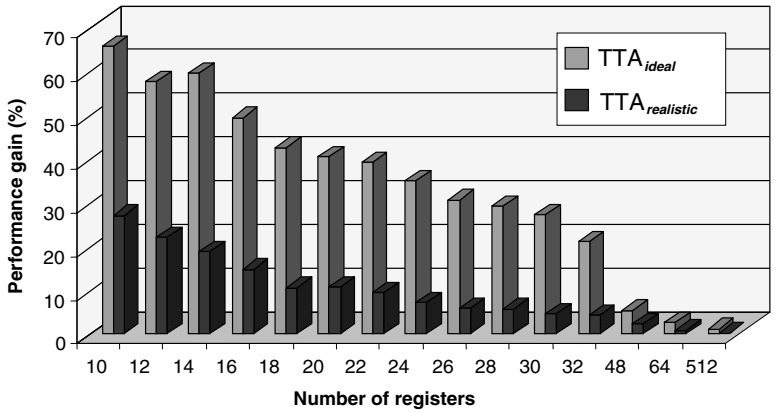


Figure 8.5: Speedup of integrated assignment compared to DCEA using software pipelining and the TTA$_{ideal}$ and TTA$_{realistic}$ templates.

8.4.2 Early vs. Integrated Assignment

It is interesting to know how integrated assignment in combination with software pipelining performs compared to DCEA. The results of Chapter 6 and Chapter 7 indicate that the performance gain of integrated assignment increases when the register pressure increases. Software pipelining increases the register pressure and thus we expected a high performance gain.

The achieved performance gain for both TTA templates is shown in Figure 8.5. As was expected, integrated assignment outperforms DCEA. When comparing the performance gain of software pipelining with the performance gain of region scheduling in Figure 7.10, an increase in performance gain can be observed. This indicates that integrated assignment indeed performs better when the register pressure is high.

The individual results of all benchmarks are listed in Appendix A. These results show that in some cases an extreme high performance gain was achieved. For example, the *rfast* benchmark when using the TTA$_{ideal}$ template with 10 registers has a speedup of 400%. Part of the speedup can be explained by the reasons mentioned in the previous chapters. Furthermore, software pipelining increases the register pressure, which makes things even worse for an early assignment approach. There are, however, other reasons.

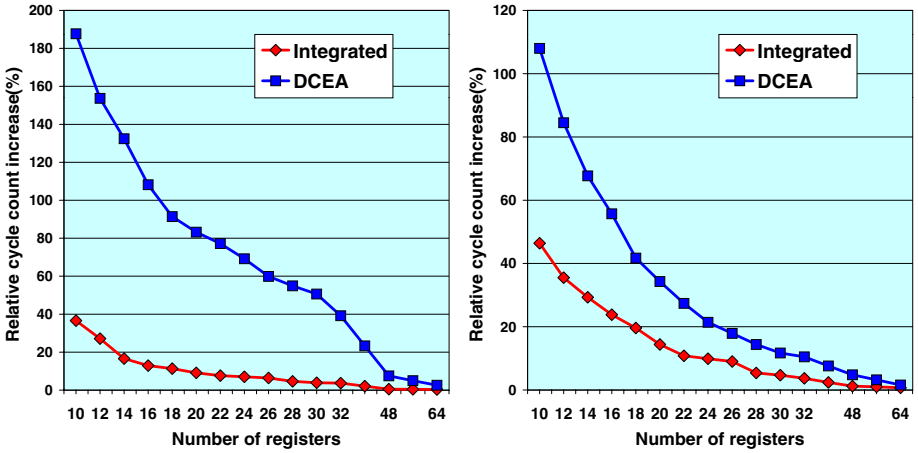
When early assignment precedes software pipelining, false dependences may hinder the generation of an optimal schedule. Not only false dependences between live ranges in the same iteration, but also false dependences between live ranges in other iterations may reduce the throughput of the loop. DCEA takes besides the intra-iteration false dependences also the inter-iteration false dependences into consideration. However, when all false dependences (in-

tra and inter) are take into consideration, the resulting interference graph has many interference edges. It is hard to determine which false dependences are really important. Therefore, it is necessary to prune the interference graph. This is accomplished by pre-software pipelining the loop without considering resource constraints and false dependences [Hoo96]. The pre-schedule is analyzed and interference edges are added to the graph IG_{fpar}. This graph contains many more interference edges than without software pipelining. DCEA has to decide which false dependences should be avoided, and which ones can be introduced when the number of registers is limited. Unfortunately, the IG_{fpar} is constructed under the assumption of an unbounded number of resources. When resources are limited, and thus the II increases, other false dependences might be more important. This effect seems to be an important reason for the large performance gain when registers are scarce.

Another reason for the performance difference is the influence of the delay lines. Delay lines require extra registers when an early assignment approach is applied. However, in the resulting code, many of the delay lines disappear because of software bypassing and dead-result move elimination. Unfortunately, an early assignment approach cannot benefit from this reduction of register requirements. Even worse, it may insert spill code for the delay lines. Integrated assignment does not have this problem because it can benefit from the TTA specific properties.

The results, shown in Figure 8.5, suggest that integrated assignment performs extremely well when registers are scarce. However, it is DCEA that performs extremely badly in these situations. This can be concluded from Figure 8.6. This figure shows the cycle count increase relative to a configuration with 512 registers. Only benchmarks with a software pipeline ratio higher than 10% are included to compute the average (see Section 4.3). The figure shows that the cycle count increase for DCEA is 187.7%, while for integrated assignment it is limited to only 36.6% when using the TTA_{ideal} template with 10 registers. For individual benchmarks this difference is even larger. For the same TTA, the benchmark *rfast* has a cycle count increase of 553.4% for DCEA, while for integrated assignment it is limited to 30.7%. It is also interesting to note that the cycle count increase of the $\text{TTA}_{realistic}$ is larger than the one for the TTA_{ideal}. Because the TTA_{ideal} has many resources, it can more easily integrated spill code in the schedule without a large performance loss.

Also some performance losses can be observed when comparing integrated assignment with DCEA (see Appendix A). As stated in [Hoo96] the performance of software pipelining heavily depends on the selection order of the operations. As proposed by Hoogerbrugge, two heuristics that define the operation selection order are tried in order to generate a valid schedule in II instructions. Because of the false dependences in early assignment, these heuristics generate different orderings for early assignment than for integrated assignment. The ordering determines whether a schedule fits into II instructions. Another ordering may result in a smaller or larger II. This effect causes the



a) TTA$_{ideal}$. b) TTA$_{realistic}$.

Figure 8.6: Average cycle count increase of integrated assignment and DCEA. Only benchmarks with a software pipeline ratio higher than 10% are included.

negative performance gain of integrated assignment for the benchmarks *instf*, *radproc* and *rfast* even when sufficient registers (512) are available. Analysis showed that these benchmarks are dominated by a small set of loops that can be software pipelined. An increase of the II with one instruction, already contributes to a performance loss of a few percentages.

The presented algorithm has a potential performance flaw that did not show up in the results. Because all instructions in the loop are taken into account when computing the interference register set, it is not possible to use the same register for two different live ranges. Software bypassing and dead-result move elimination may have obscured this effect. Furthermore, the loops considered were relatively small. When larger loops are taken into consideration, which results in a larger kernel, this effect may become more visible. Therefore, in the future the algorithm should be extended in order to allow the re-usage of registers in software pipelined loops with a large II.

8.5 Conclusions

In this chapter, the relation between register assignment and software pipelining is discussed. It was shown that software pipelining increases the register pressure. Based on the work as described in the Chapters 6 and 7, we proposed a method that integrates register assignment and software pipelining. We also addressed the problem of register shortage. We have shown that the new de-

veloped method can handle this by either increasing the II, or by inserting spill code. The experiments showed that increasing the II in some situations slightly improves the performance. On average, however, the impact of increasing the II is practically non-existing. Spilling is the better choice.

The performance of the proposed method is compared with DCEA. The experiments showed that integrated assignment outperformed DCEA. DCEA suffers from the increased register pressure in software pipelined loops. Its inability to exploit software bypassing and dead-result move elimination, and the used method to predict false dependences heavily contribute to its performance penalty. The impact of DCEA on the cycle count increase leads to the conclusion that the combination of DCEA and software pipelining is not a good choice when registers are scarce.

The Partitioned Register File

9

The general-purpose registers of a microprocessor are located in the register file (RF). These registers can be accessed through a limited number of RF-ports. Reading N values in parallel requires N read ports on the RF. An architecture that exploits instruction-level parallelism (ILP) should be able to read and write multiple register values concurrently; ILP architectures therefore require multi-ported register files. Many ILP based processors have a single multi-ported register file that is commonly known as a *monolithic* multi-ported register file. Unfortunately, a monolithic multi-ported RF increases chip area [CDN95], causes extra delay [WRP92] and boosts power consumption [ZK98]. Therefore, an alternative RF organization is needed in order to exploit large amounts of ILP.

This chapter is organized as follows. Section 9.1 describes the problems of a monolithic multi-ported RF: chip area, access time and power consumption are addressed. To overcome these problems, an alternative RF organization is discussed: *the partitioned RF*. Although a partitioned RF has hardware advantages, from a code generation point of view, it introduces some complications. To avoid large performance penalties, the compiler has to take the partitioning into consideration. In particular, the register allocator must be adapted; it has to distribute the variables over the partitioned RF. In Section 9.2 partitioning and early assignment is discussed. The consequences for late assignment are discussed in Section 9.3, and how integrated assignment handles partitioned RFs is discussed in Section 9.4. In addition, related work is addressed in these sections. Finally, Section 9.5 concludes this chapter.

9.1 Register Files

To increase performance, modern microprocessors execute many operations in parallel. To allow the exploitation of a high degree of ILP, RFs with a large number of registers are needed. For example, the Itanium processor and the TM1000 contain 128 integer registers each [Abe00, HA99]. Parallel execution of multiple operations also requires simultaneously reading and writing of multiple values from and to the RF. This is done by means of the RF-ports. In our formulation, each RF-port is either a read port or a write port and hence operands compete for RF-port usage within the type of access (read operations conflict with read operations for accessing read ports, and vice versa for write operations).

VLIWs require $3K$ RF-ports assuming K FUs (see Section 2.1). The inputs and outputs of each FU have a connection to the RF. From the compiler's point of view, this interconnection is beneficial because it provides an orthogonal programming model. As a result, a standard register assignment technique can be used, which in turn, produces high performance code. Unfortunately, the high number of RF-ports needed for VLIWs is a major problem when the number of FUs increases. E.g., a VLIW with 5 FUs requires 15 RF-ports. Even if such a large multi-ported RF could be built, it is likely to introduce performance degradation because of an increased overall cycle time [Jol91]. This degradation may greatly affect the benefit of the complete interconnection to a single RF and reduces the theoretical performance achievable with many FUs.

TTAs do not have a direct relation between the number of FUs and the number of RF-ports. This allows the design of microprocessors with a high number of FUs, while the number of RF-ports is moderate. However, the number of RF-ports must be large enough to support the available ILP. Another factor, which reduces the pressure on the RF-ports, is the ability to software bypass results directly from the producing FU to the consuming FU. This saves the use of a read port. When dead-result move elimination can be applied, also the use of one write port and a register is saved. In [HC94], it is shown that TTAs can reduce the RF-port requirements by 50% or more compared to VLIWs.

It is to be expected that the ongoing research in compiler and microprocessor technology will make it feasible to execute 16 [LW97, RJSS97, SCD+97] or more operations in parallel. Although the number of RF-ports for a TTA are smaller than for a VLIW (A VLIW would require 48 RF-ports) to achieve the same performance, the number of RF-ports must still be substantial to exploit the increase in computing power.

Silicon area, access time and power consumption increase when the number of ports and registers of an RF increase. In this section, models from literature are presented to model silicon area (Section 9.1.1), access time 9.1.2 and power consumption 9.1.3. We do not claim an exhaustive analysis, which would be beyond the scope of this thesis. In Section 9.1.4 an alternative RF architecture is presented. This RF architecture solves the problems of a large monolithic RF.

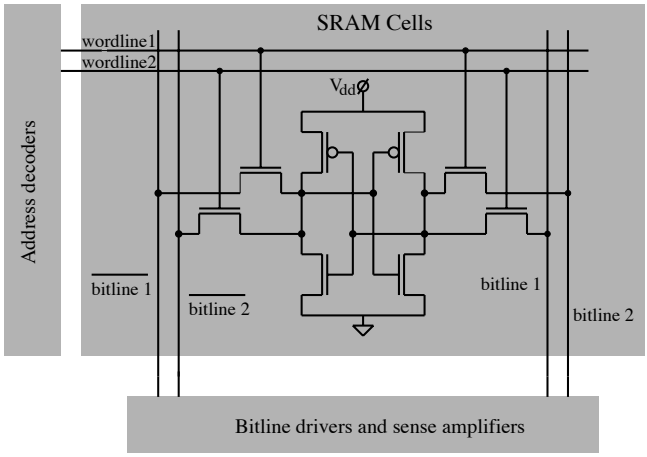


Figure 9.1: Implementation of a 2-ported RF (only one bit has been drawn).

9.1.1 Silicon Area

The area of a multi-ported RF is largely dependent on the number of registers, the width of the registers (i.e. the number of bits per registers), the number of RF-ports and the routing (wiring) area. A register in an RF consists of a number of cells. Figure 9.1 [vdG98, JC95] shows a 1-bit, 4-transistor SRAM cell with two ports. An RF with 24 32-bit registers will contain 32 of those cells in a row (forming one word or register) and 24 registers in a column.

In the worst case, each cell must be able to drive all the read ports simultaneously. To accomplish this, the area of the two inverters must be increased to drive the increased load. In [CDN95], it is claimed that their area grows linear with the number of registers. Another factor that influences the area of an RF is routing. According to [CDN94, CDN95, Cor98] the area for routing grows quadratically with the number of RF-ports. In [CDN94] a large number of published RF designs is studied. This study and the study of Corporaal [Cor98] indicated that the silicon area is usually dominated by the routing area. Even for a limited number of RF-ports the cell layout is almost completely occupied by bitlines and wordlines. The area model used in this thesis is based on the model of [CDN94]:

$$Area_{RF} \sim R_{RF}(1 + \delta_a \cdot N_{ports})^2 \tag{9.1}$$

where R_{RF} is the number of registers, N_{ports} the number of RF-ports and δ_a the percentage dimensional increase determined by each RF-port. According to [CDN95], δ_a is in the range 0.05 to 0.20 for most of the considered designs. This value depends on the number of wiring levels. Note, that the model does not give an absolute area measure, but the relation between various RF con-

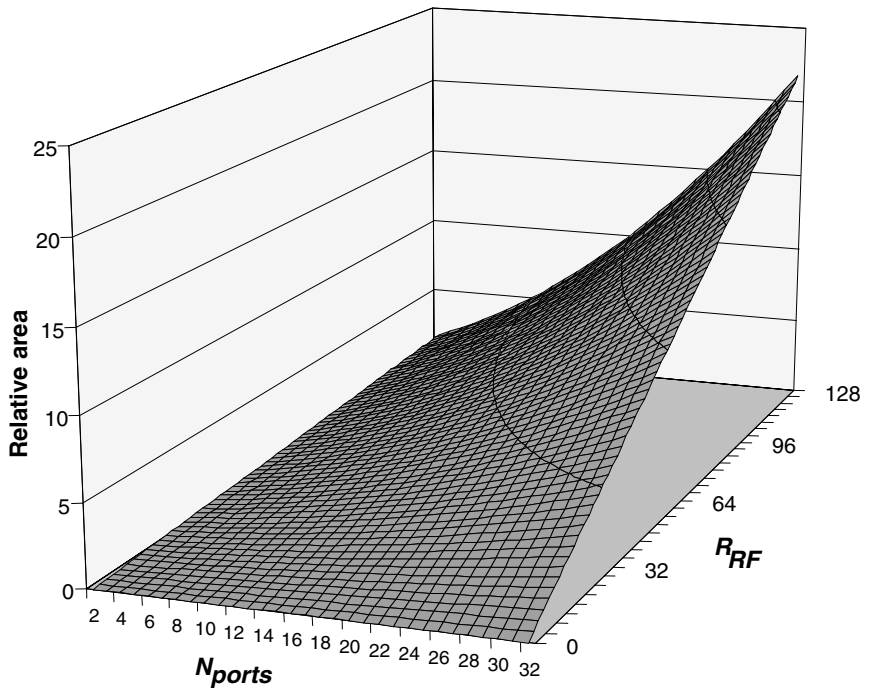


Figure 9.2: Area of various monolithic RF configurations relative to the area of a monolithic 8-ported RF with 32 registers assuming $\delta_a = 0.1$.

figurations. Figure 9.2 shows the relative area as a function of the number of RF-ports and registers assuming $\delta_a = 0.1$.

9.1.2 Access Time

In current microprocessor designs a trade-off has to be made between size, bandwidth and speed of the RF [CDN94]. Simultaneous access to all RF-ports requires appropriate sizing of the drivers. This results in an increased cell dimension because the inverters must be sized to drive all the lines in all situations. Also the line length increases and hence results in an increased propagation delay. The access time of an RF can, according to [CDN95] and [Cor98], be modeled as a linear function of the number of RF-ports. From the results presented in [FJC95], it follows that the access time is also proportional with the number of registers. Based on these observations we model the relative access time increase for a multi-ported RF as:

$$T_{access} \sim 1 + \rho_r \cdot R_{RF} + \rho_p \cdot N_{ports} \tag{9.2}$$

where ρ_r is the percentage cost increase per register and ρ_p gives the percentage cost increase per RF-port. In [CDN95] a value of 0.1 for ρ_p is suggested. Farkas et al. [FJC95] observed that the access time of an RF is far more affected by a doubling of RF-ports than a doubling of the number of registers. Based on their results we use $\rho_r = 0.01$.

In future microprocessor designs, the number of registers and RF-ports will grow. The above observations about the relation between access time and RF-ports make the RF a fundamental design issue for large TTA, VLIW and superscalar processors [Cor98, CDN95, FJC95, EFK+98, CGVT00].

9.1.3 Power Consumption

The last discussed parameter is power consumption. When taking into account the growth of both the storage size and the number of ports, it is to be expected that the power portion of a multi-ported RF will grow in the future. In [ZK98], the authors have described and analyzed the energy complexity of multi-ported RFs. Their study showed that the average access energy is proportional with the number of RF-ports and with the number of registers.

$$E_{RF} \sim (c_1 + c_2 \cdot R_{RF})(c_3 + c_4 \cdot N_{ports}) \tag{9.3}$$

The constants c_1, c_2, c_3 and c_4 can be estimated from the figures in [ZK98] for 0.5 μm technology using CMOS. The result is shown in Figure 9.3. Furthermore, in [ZK00] it was shown that the power consumption of an RF in a superscalar or VLIW processor can be described as $P \sim (IW)^{1.8}$, where IW is the issue width. In a properly designed superscalar or VLIW processor, an increase in issue width has as a consequence that the number of RF-ports increases as well as the number of registers. According to Zyuban and Kogge, the RF is among the components with the highest energy growth when the issue width increases. This leads to the conclusion that an increase in issue width (and thus exploitable ILP) results in an almost quadratical increase in power consumption. Zyuban and Kogge [ZK98, ZK00] conclude that none of the known circuit techniques solves the problem of rapid RF power growth for microprocessors with increasing ILP. Also aggressive technology scaling does not solve the problem. They suggest to develop new alternatives for the monolithic RF.

9.1.4 Partitioned Register Files

When designing microprocessors that can exploit a large amount of ILP an alternative for the large monolithic RF must be found. A frequently used technique is to split the set of registers into two banks (RFs), one for the integer and the other for the floating-point registers. However, this is only a partly solution since the number of ports and registers for each RF is still large. In addition to dividing register types over different RFs, one can also choose to split the monolithic RF per type into small RFs. This results in either a distributed or

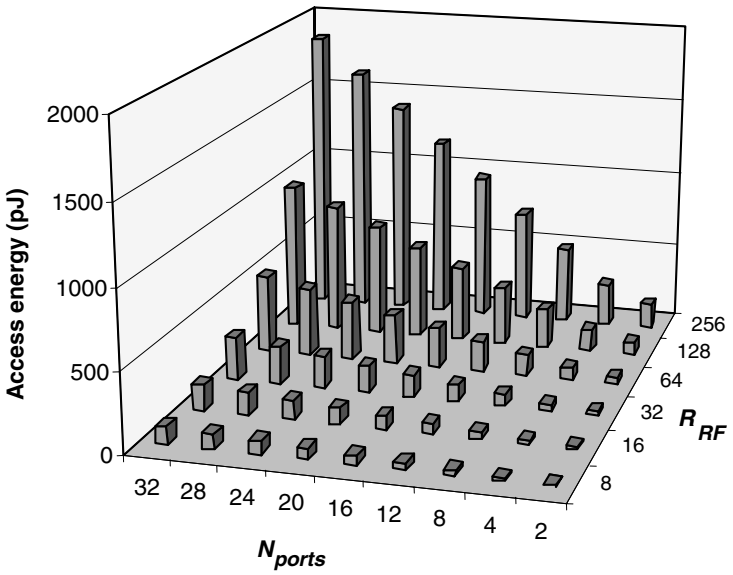
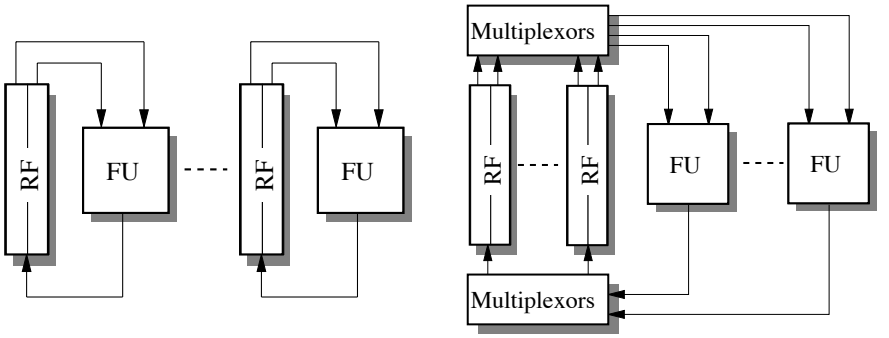


Figure 9.3: Average access energy per instruction.

a partitioned register file configuration [Lee92]; these configurations are shown in Figure 9.4.

In a distributed RF configuration each FU, or cluster of FUs, has direct access to one RF only. Access to other RFs requires register copy operations or complex bypassing logic. This architecture, also known as *multicluster architecture* [FCJV97], has been used in the Multiflow TRACE architecture [SS93], the vector processor CM-5 of Thinking Machines and DEC's Alpha 21264. The Alpha 21264 has two integer FU clusters, each with a separate RF. It was decided



a) Distributed RF. b) Partitioned RF.

Figure 9.4: Dividing an RF.

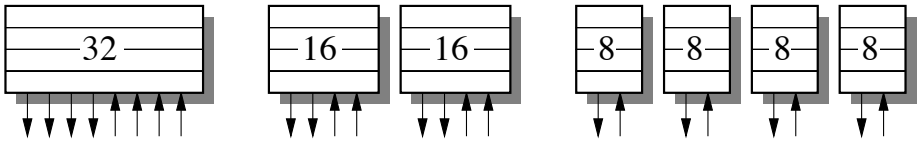
to partition the RF because it was indeed in the critical timing path [Gwe96]. In the VLIW family TMS320C6000 the idea of a distributed RF is used also. The devices of this family are based on the VelociTI core, which has two clusters, each with a local RF and four FUs.

The idea of a distributed RF is also applied in the Multiscalar architecture [SBV95]. The key idea behind this architecture is to split one wide issue processing unit into multiple narrow-issue processing units. Each narrow-issue unit consists of FUs and an RF. The trace processor [RJSS97] is based on the Multiscalar idea as well. Like Multiscalar, trace processors are proposed to overcome the architectural limitations on ILP of superscalar processors. Trace processors exploit the characteristics of traces. A trace processor consists of multiple processor elements that each have the organization of a small superscalar processor. Trace data characteristics, local versus global values suggest a hierarchical RF implementation: a local RF per trace for holding values produced and consumed solely within a trace, and a global RF for holding values that are live between traces. The local RFs can be regarded as a distributed RF, while the global RF is used to copy the results between processing elements.

As shown by Capitanio [CDN93], compilation for a distributed RF configuration is difficult and may result in a large performance loss (up to 75%). Note further, that the register copy operations between RFs may require extra ports on the RFs. These ports are not shown in Figure 9.4a.

A different approach is to partition the RF as shown in Figure 9.4b. Although a partitioned RF provides less connectivity between FUs and registers, when compared to a monolithic RF, each FU still has direct access (for reads and writes) to any register. Therefore, no extra register copies are required. However, still performance losses are to be expected because the limited number of ports per RF may lead to *RF-port conflicts*, i.e. when two or more accesses need the same port in the same cycle.

Partitioned RFs have been used for many years in vector processors. Each vector register in a vector processor can be regarded as a separate RF with only a limited number (often one) read and write ports [Lee92]. Partitioning can also be compared to the use of multiple banks; e.g. a cache split into two banks, each having only one port, can handle two accesses at a time if these accesses are not to the same bank. In both cases, the hardware detects and handles the access conflicts. The Motorola 56000 and the NEC 77016 allow multiple concurrent access to different on-chip memory banks. To exploit this feature, applications are hand-written. In [SM95], a compiler technique is described, which can generate code for these types of architectures. This technique uses graph coloring to assign registers as well as memory references to memory banks during late register assignment. Besides register conflict edges, also memory bank conflict edges and edges that reflect architectural imposed constraints are added. Associated with each edge is a cost. Simulated annealing is used to minimize the total cost. The total cost is defined as the sum of the costs of all edges whose constraints are not met.



a) Monolithic RF. b) 2-partitioned RF. c) 4-partitioned RF.

Figure 9.5: Register file configurations.

In this chapter, we propose the use of a partitioned RF, for which access conflicts are handled (i.e. avoided) by the compiler. A partitioned RF fits very well in the TTA template. No extra hardware facilities are required. Examples of some partitions are given in Figure 9.5.

Figure 9.6 draws the area for three RF configurations, a monolithic RF, one with two partitions, and one with four partitions in units of an 8-ported monolithic RF with 32 registers (i.e. relative to the configuration of Figure 9.5a). Each configuration has a total of 32 registers. The registers and RF-ports are equally divided between the new partitions. From the figure, it can be clearly seen that the area occupied by a partitioned RF is significantly smaller than the area of the monolithic RF.

In the remainder of this chapter, the RF configurations of Figure 9.5 are used. Based on the observations made in this section, the following conclusions can be drawn:

- A monolithic 8-ported RF with 32 registers has an area 1.65 times the area of its two times partitioned counterpart, and an area 2.25 times the area of its four times partitioned counterpart.
- A monolithic 8-ported RF with 32 registers has an access time that is 1.36, respectively 1.66 times, the access time of its two, respectively four times partitioned counterpart.
- The average access energy per instruction for a monolithic RF is 1.53, respectively 1.96 times, the average access energy of its two, respectively four times partitioned counterpart.

Partitioning the RF has advantages: less chip area, lower access times and a lower power consumption. However, partitioning also has a drawback. In a single monolithic RF, each register can be accessed via each RF-port. In a partitioned RF, this is no longer valid. The consequences for code generation are discussed in the remainder of this chapter.

9.2 Early Assignment and Partitioned Register Files

Partitioning the RF does not induce any changes to the TTA post-pass scheduler. Based on the index of a register, it selects an RF and an RF-port of the

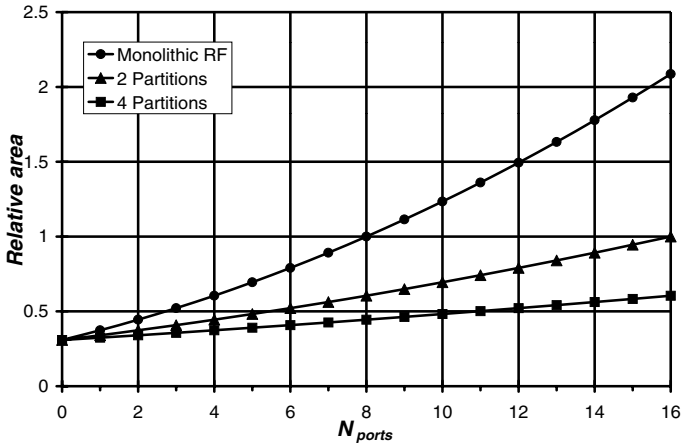


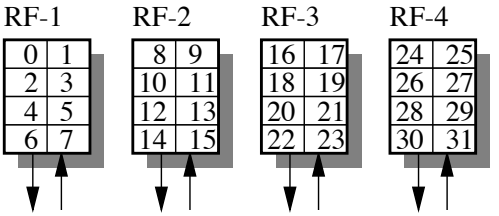
Figure 9.6: Area of various 32 register RF configurations relative to the area of a monolithic 8-ported RF with 32 registers. Model based on Equation 9.1 with $\delta_a = 0.1$.

correct RF partition. The register allocator is responsible for assigning registers to variables and thus determines in which RF partition the variable will reside. Distribution of the variables over the RFs should be done with care in order to reduce the performance penalty imposed by the partitioning. In this section, four distribution methods in combination with early assignment are described and evaluated. The first two methods use a simple distribution method, while the latter two are based on more advanced heuristics.

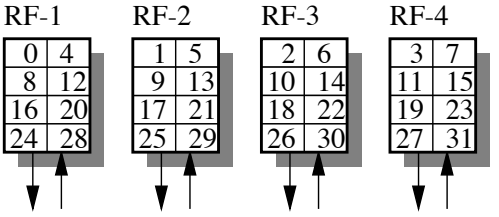
9.2.1 Simple Distribution Methods

Let us assume that all registers fit into a single address space; e.g. if there are four RFs with 8 registers each, then we assume that we can address each register with an address between 0 and 31. The problem is how to distribute the register addresses over the RFs. Two distribution methods are considered: the *vertical* and the *horizontal* distribution, see Figure 9.7. The vertical distribution assigns the register addresses 0..7 to RF-1, 8..15 to RF-2 and so on. The horizontal distribution assigns the register addresses in a round robin fashion to the RFs as shown in Figure 9.7b.

First, experiments were conducted with the vertical distribution. The $\text{TTA}_{realistic}$ template as described in Section 4.2 is used in combination with the RF configurations of Figure 9.5. In addition, a two and four partitioning of an RF with 512 registers and four read and four write ports was included in the experiments. Figure 9.8 shows the performance *decrease* of the two and four partitionings relative to the performance of the monolithic RFs when using region scheduling. The performance loss for a partitioning in four parts is quite



a) Vertical distribution.



b) Horizontal distribution.

Figure 9.7: Distributions of register addresses over a four times partitioned RF.

large, especially for the 4x128 partitioning. This is because the register allocator picks the first available register, and therefore registers with a low index in the register address space are picked relatively often. These registers are, however, located in the first RF and must share the same number of ports. This results in many access conflicts during the scheduling process. It is interesting to note that the relative performance penalty is lower when the partitioned RF contains 32 instead of 512 registers. For these smaller RFs, the register allocator has to use multiple partitions in order to avoid spilling, while for the RF with 512 registers a single partition may be sufficient. However, the number of RF-

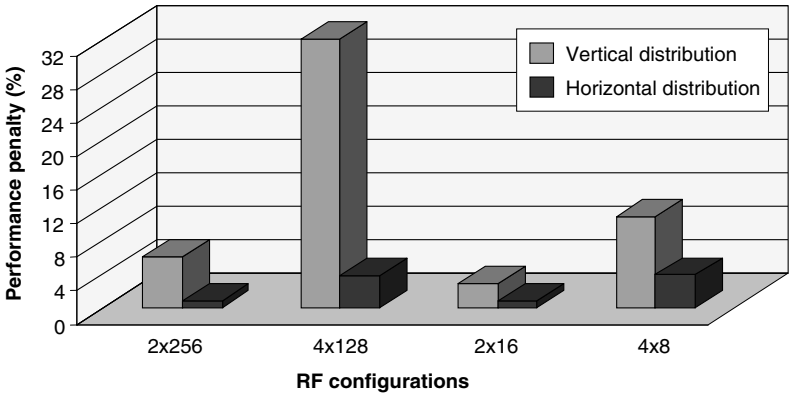


Figure 9.8: Results of the vertical and horizontal distribution for various RF partitionings.

ports on a single partition is small and consequently, the number of RF-port conflicts increases, which results in a larger performance penalty.

The *horizontal distribution* distributes the register address such that the number of accesses to the RFs are more uniformly distributed. The relative performance loss when applying this distribution is also shown in Figure 9.8. The results show that the horizontal distribution performs surprisingly good, and much better than the vertical distribution. The vertical distribution resulted in an average performance loss of more than 35.6% for the 4x128 partitioning while the horizontal distribution limits the performance loss to only 3.8%. The results per benchmark are given in Appendix B.

9.2.2 Advanced Distribution Methods

Both the vertical and horizontal distribution do not consider the impact of simultaneous accesses to an RF. When transports are independent, the scheduler may schedule them in the same instruction. However, the scheduler cannot schedule them in the same instruction if there are more transports from or to the same RF than there are ports on that RF. The scheduler has to delay one or more transports, which may reduce the performance. In the remainder of this section, two strategies are proposed that try to avoid RF-port conflicts by mapping variables, whose transports can be scheduled in the same instruction, onto different RFs.

Definition 9.1 *The RF-port interference graph $IG_{\text{RF-ports}} = (N_{var}, E_{\text{RF-port}})$ is a finite undirected graph, with N_{var} the set of variables and $E_{\text{RF-port}}$ the set of potential RF-port conflicts. These edges are augmented with a weight $W_{\text{RF-port}}$, which reflects the importance of avoiding a particular RF-port conflict.*

The two proposed strategies differ in the way $IG_{\text{RF-ports}}$ is constructed and the weights associated with the edges.

To select a register r_i for a variable v_i, the information from the forward parallel interference graph IG_{fpar} and the RF-port interference graph $IG_{\text{RF-ports}}$ is used. A register is selected which respects all interferences in both graphs. When no such register is available, the register allocator selects a register that minimizes the following function:

$$\begin{aligned} C_{Total}(r_i, v_i) \;=\; & W_1 \cdot \sum_{v_j \in N_{var} \wedge r(v_j) = r_i} W_{\text{ffdp}}(v_i, v_j) \\ & + W_2 \cdot \sum_{v_j \in N_{var} \wedge r(v_j) \in RF(r_i)} W_{\text{RF-port}}(v_i, v_j) \\ & + W_3 \cdot C_{\text{state preserving}}(r_i, v_i) \end{aligned} \tag{9.4}$$

where

$$C_{\text{state preserving}}(r_i, v_i) = \begin{cases} C_{\text{caller-saved}}(v_i) & : \quad r_i \in R_{\text{caller-saved}} \\ C_{\text{callee-saved}}(v_i) & : \quad r_i \in R_{\text{callee-saved}} \end{cases}$$

and $RF(r_i)$ is the register file of register r_i. The constants W_1, W_2 and W_3 represent the weight of each of the three costs. These weights can be used to tune the algorithm. Initially, they are equal. When $C_{Total} \neq 0$ the register allocator has to decide whether to spill a variable, to introduce false dependences or to introduce RF-port conflicts. In our implementation the register allocator always tries to solve this problem by introducing a false dependence, a RF-port conflict or a combination of both conflicts.

Data Independence Heuristic

The RF-port interference graph, when using the *data independence heuristic*, is constructed by inspecting all pairs of independent transports in the DDG. The following edges are added to the set $E_{RF\text{-}port}$:

- (v_i, v_j) : if one transport defines v_i and the other defines v_j.
- (v_i, v_j) : if one transport uses v_i and the other uses v_j.

The weight $W_{RF\text{-}port}$ is calculated in the same manner as is done for the edges in the forward false dependence graph, see Equation 5.1.

The results when using this heuristic were disappointing. On average, the performance loss due to RF partitioning is increased compared to the horizontal distribution. The larger performance loss is most likely caused by the fact that the data independence heuristic avoids many RF-port conflicts, which do not show up in the generated schedule. Because the weights $W_{RF\text{-}port}$ are included in the register selection heuristic, some unwanted false dependences were introduced. Decreasing the weight W_2 to one tenth of the other weights did result in a small performance improvement; however, the results of the horizontal distribution could not be met. The results per benchmark are shown in Appendix B.

Intra-Operation Heuristic

The results of the data independence heuristic show that the number of edges in $E_{RF\text{-}port}$ should be reduced. The *intra-operation heuristic* tries to avoid multiple reads within a single operation from the same port-limited RF. We will illustrate this problem with an example: suppose an RF has only one read port. An operation usually needs two operands. If both operands are located in the same RF then it is impossible for the scheduler to schedule the two transports in the same instruction. If, however, the operands are located in different RFs, their transports can be scheduled in the same instruction and hence the performance improves. This heuristic constructs the $IG_{RF\text{-}ports}$ by inspecting all operations with multiple operands. For each combination of operand variables an edge is added to $E_{RF\text{-}port}$. However, an edge is only added when the live range has multiple uses. This restriction is added because it is very likely that a live range with a single use is eliminated from the schedule by dead-result move

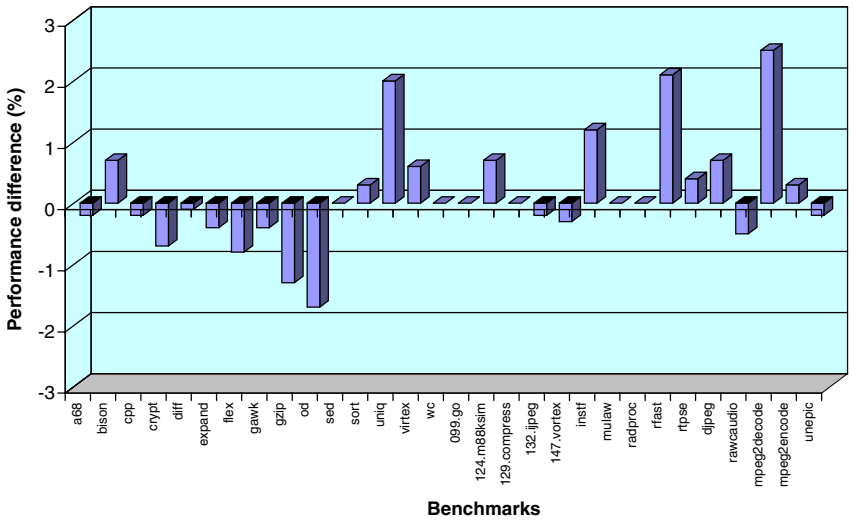


Figure 9.9: Performance comparison of the horizontal distribution and the intra-operation heuristic for the 4x8 partitioning.

elimination and thus never causes an RF-port conflict[1]. To each edge (v_i, v_j) a weight $W_{RF\text{-}port}$ is added. This weight is equal to the execution frequency of the operation's basic block.

The intra-operation heuristic turned out to be better than the data independence heuristic. On average, the results of the intra-operation heuristic are equal to the results of the horizontal distribution. However, per benchmark the results vary, see Appendix B. Figure 9.9 shows the performance differences per benchmark for the 4x8 RF partitioning. A positive value in this graph indicates that the horizontal distribution performs better. These results show that the performance of the horizontal distribution can be improved, but that no heuristic was found that on average resulted in a better performance.

9.2.3 Equal Area Compiling

One could argue that the area saved by RF partitioning can be used for other resources. Let us investigate what happens when the saved area is spend on extra registers. Table 9.1 lists the new equal area partitioning of a monolithic RF with 32 registers, four read, and four write ports. In the experiments, the horizontal distribution was used; with this heuristic the best results were obtained. Figure 9.10 shows the performance penalties of these new partitionings. From the figure we conclude that with equal area compiling the performance losses caused by RF partitioning can be turned into a performance gain. For some

[1]Note, that the inspected pairs in the intra-operation heuristic are a subset of the inspected pairs in the data independence heuristic.

Table 9.1: RF configurations.

RF configuration	Partitions	R_{RF}	$N_{ports}(read)$ per partition	$N_{ports}(write)$ per partition
2x26	2	26	2	2
4x18	4	18	1	1

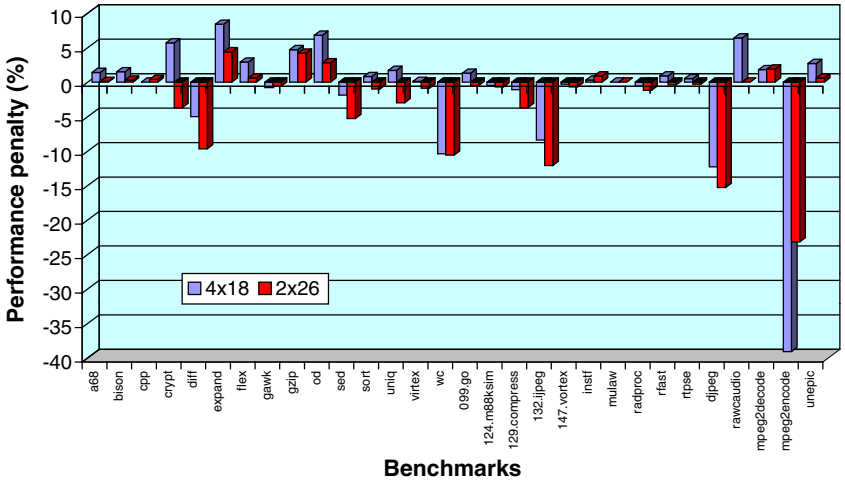


Figure 9.10: Results of equal area compiling.

benchmarks the performance gain is large, for example *mpeg2encode*. Because more registers are available, less spill code is required and thus the performance increases. Other benchmarks show disappointing results. Because the RFs in the new RF configurations contain more registers than the original RFs, potentially more RF-port conflicts can arise. This may result in a performance penalty. A register assignment strategy that divides the variables evenly across the RFs, such as the horizontal distribution, may reduce the impact of having more registers behind a small number of RF-ports. The results of many benchmarks are almost not affected by adding extra registers. These benchmarks do not require more than 32 registers. Adding more registers will not help to gain performance. For these benchmarks it is more profitable to spend the saved area on other resources like FUs, a larger cache, etc.

9.3 Late Assignment and Partitioned Register Files

In Chapter 5, we discussed why late register assignment is not an attractive solution for TTAs. In this section, the consequences when applying late assign-

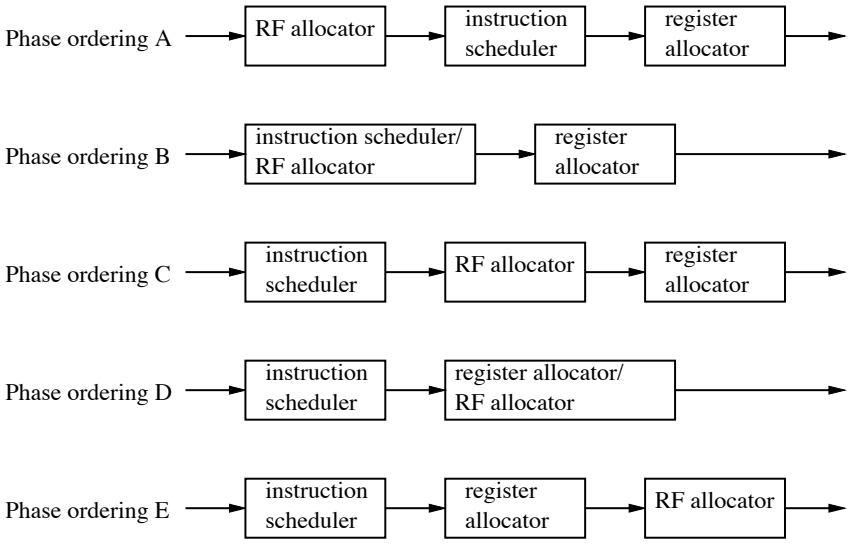


Figure 9.11: Late assignment phase orderings in the context of a partitioned RF.

ment in combination with a partitioned RF are discussed. Allocating variables to RF partitions can be applied in the instruction scheduling phase, the register assignment phase or in a separate phase. All phase orderings are given in Figure 9.11 and are shortly addressed[2].

Phase ordering A: Early RF assignment
When assigning variables to RFs before the instruction scheduling and register assignment phase, both phases are constrained by the RF assignment. Due to RF-port conflicts, the instruction scheduler has to delay some operations. This results in less efficient code. Also the freedom of the register allocator is restricted; it can only map a variable onto a register in the assigned RF. Depending on the partitioning algorithm, some RFs are over-utilized, there are too many variables for the available registers, while others have a surplus on registers. For the variables in the over-utilized RF, the post-pass register allocator has two spill options: (1) spilling to memory or (2) spilling to an under-utilized RF with the use of copy operations. This is only possible when enough RF-ports are available. Both approaches degrade the performance.

Phase ordering B: Integrated scheduling and RF assignment
In this phase ordering, the instruction scheduler maps variables to RFs. The RFs are assigned in such a manner that the performance is maximized from the instruction scheduler's point of view. The scheduler assumes that all the

[2]For early assignment also several phase orderings can be considered, however, most of them are senseless since the register assignment determines the RF.

RFs are infinitely large. The RF assignment may hinder the register allocator; it can only map a variable onto a register in the RF assigned by the scheduler. In the worst case, the scheduler assigns all variables to one RF, while the other ones are empty. Typically the variables will be scattered over all RFs, however, the distribution of the variables over the RFs may not be optimal for register assignment. As in phase ordering A, the register allocator has to insert spill code or extra copy operations.

Phase ordering C: In between RF assignment
In this phase ordering, the instruction scheduler assumes a single infinitely large RF. Consequently, it will likely generate a high performance schedule. It is, however, not guaranteed that this schedule is correct when using a partitioned RF. The RF assignment phase assigns the variables to RFs and inserts, when needed extra instructions to correct the schedule. The register allocator is responsible for allocating the variables to the registers of the RFs.

In literature, two approaches are described that perform RF partitioning after instruction scheduling. In [CDN93], the RF allocator partitions the operations in already scheduled VLIW instructions into a number of substreams equal to the number of RFs. The proposed method partitions the operations in such a way that it minimizes the amount of communication between the substreams. It is not very likely that a clean partitioning can be found, i.e., without any data movement between the substreams. When communication is necessary, data movement operations are inserted and the code is locally rescheduled. This method only addresses the distribution of variables across RFs. It does not address register assignment; it is assumed that each RF has sufficient registers to hold all variables. Furthermore, the method is only applied to straight-line code (loop bodies). The reported performance losses range from 4% to 40% for two partitions and from 15% to 75% for four partitions.

In [CND95], a hypergraph-based coloring method is proposed to model competition among variables for RF-ports on a VLIW with a partitioned RF. The nodes in such a hypergraph represent multichains. A multichain is a collection of scheduled operations that define or use the same variable (similar to du-chains without a direction of the edges). The hyperedges between the nodes represent the interferences among multichains. Two types of hyperedges are defined: RF-port conflicts for reading and RF-port conflicts for writing a variable. Each hyperedge connects sets of multichains, which compete for an RF read/write port in the same instruction. Graph coloring is used to assign multichains to RFs. The colors represent the RFs. A hyperedge should not connect more nodes with the same color than there are RF-ports. Not always a legal coloring can be found. In these situations, operations are moved to other instructions and copy operations are inserted. The presented results show that only a small performance degradation has to be accepted, lower than 5%. However, no assumption about the size of the RFs is made. This may lead, as in [CDN93], to an unevenly distribution of variables to RFs and may even lead to unnecessary spilling when the RFs have a limited number of registers.

Phase ordering D: Integrated register and RF assignment
In this approach, the register allocator is responsible for assigning RFs to variables. Prior to register assignment the instruction scheduler places the operations or transports in the instructions while respecting the total number of RF-ports. However, the scheduler does not assign RF-ports to operations; this is the responsibility of the register allocator. The register allocator cannot assign registers in such a way that more writes or reads access simultaneously the same RF than there are write respectively read ports. It is not likely that this restriction is always respected by the instruction scheduled, and thus rescheduling is required. Furthermore, the register allocator has to insert spill code when needed.

Phase ordering E: Late RF assignment
In this phase ordering both instruction scheduling and register assignment assume a single RF. During RF assignment, the registers are mapped onto RFs. The assignment of the registers combined with the already generated schedule most likely results in RF-port conflicts. When no legal RF assignment can be found, some operations must be delayed and rescheduled to reduce RF-port conflicts.

All mentioned phase orderings have their advantages and drawbacks. Since not much research is done in this area, it is hard to say which phase ordering outperforms the other. Although in [CND95] good results are reported, they did not take into consideration the size of the RFs. Furthermore, all methods require rescheduling of already scheduled code. As was observed in Section 5.2, this is complex in the context of TTAs.

9.4 Integrated Assignment and Partitioned Register Files

The separation of register assignment, instruction scheduling and RF-assignment in different phases has its drawbacks. Decisions that seem to be advantageous in an earlier phase can have a negative effect on the total performance. In early assignment, the register allocator has no idea which assignments will result in RF-port conflicts in the instruction scheduling phase; only assumptions about potential conflicts can be taken into considerations. An integrated approach has more knowledge about RF-port availability per instruction. Because the capacity of the RF is taken into consideration as well, the RFs are not over-utilized such as with late assignment. This reduces the amount of spill code. Furthermore, no rescheduling is required.

Despite these advantages, almost no research is done in the area of integrated assignment and partitioned RFs. To the best of our knowledge, only in the Bulldog compiler [Ell86] integrated assignment is addressed in the context of an instruction-based list scheduler for clustered VLIWs. The proposed method

```
add r6, r2, r4
```

a) Problematic operation.

```
move r2, r3
add  r6, r3, r4
```

b) Solution using copy insertion.

Figure 9.12: Inserting a copy operation. It is assumed that the register set is divided over two RF, one with the even registers and one with the odd registers.

selects an RF which has (1) free registers, (2) can be reached from the producing FU and (3) has the fewest RF-port accesses in the instruction in which the new access will be scheduled. No attempt is made to include possible conflicting future references (that is later in the schedule) into the selection method. Unfortunately, no measurements are provided to evaluate the performance decrease due to a partitioned RF. Ellis also addressed a typical OTA related problem in relation to partitioned RFs. Many operations require two operands, which have to be read in the same instruction. When both operands are located into the same RF, which has only one RF-port, the operation cannot be executed. Inserting a copy operation into the code can solve this problem. An example is given in Figure 9.12. Figure 9.12a shows the original operation. This operation cannot be scheduled since both operands must be read simultaneously from the same single read ported RF. The principle of copy insertion is demonstrated in Figure 9.12b. Note that this is not a problem for TTAs, since the operands do not have to be read in the same instruction.

In the Chapters 6, 7 and 8, we introduced a new integrated assignment approach. In the following, we extend our integrated assignment approach in such a way that it can efficiently handle a partitioned RF.

9.4.1 Local Heuristics

Recall that our integrated approach assigns a register to a variable when the operation that first references this variable is scheduled. At that moment, it also decides in which RF the variable will reside. Algorithm 6.1 is changed to Algorithm 9.1 in order to support the use of partitioned RFs[3]. The main difference is that now the functions SELECTSRCREGISTER and SELECTDSTREGISTER have to select a register from the RF that is connected to socket si respectively di. This new algorithm implements a *first-fit* approach. The RF whose sockets are in the front of the socket list is tried first and thus will be used more extensively than the other RFs. Only when no sockets are available, or when

[3]During region scheduling registers are unassigned to facilitate importing. Because some of the accesses of the associated variable are already scheduled and bound to an RF, the register allocator has to assign a register from the same RF as the original register. This is accomplished by adding extra information to the move that indicates whether the choice of RFs is free or not. This information is used in the functions AVAILABLEORDEREDSRCSOCKETS and AVAILABLEORDEREDDSTSOCKETS, which only return the sockets of the correct RF.

Algorithm 9.1 ASSIGNTRANSPORTRESOURCES(m, i)

```
src = SOURCE(m)
dst = DESTINATION(m)
FOR EACH si ∈ AVAILABLEORDEREDSRCSOCKETS(src, i) DO
  FOR EACH di ∈ AVAILABLEORDEREDDSTSOCKETS(dst, i) ∧ di ≠ si DO
    FOR EACH mb ∈ AVAILABLEMOVEBUSES(si, di, i) DO
      IF ISVARIABLE(src) THEN
        r_src = SELECTSRCREGISTER(m, i, RFCONNECTEDTO(si))
        IF r_src = ∅ THEN
          continue
        ENDIF
      ENDIF
      IF ISVARIABLE(dst) THEN
        r_dst = SELECTDSTREGISTER(m, i, RFCONNECTEDTO(di))
        IF r_dst = ∅ THEN
          continue
        ENDIF
      ENDIF
      IF ISVARIABLE(src) THEN
        ASSIGNREGISTER(src, r_src)
      ENDIF
      IF ISVARIABLE(dst) THEN
        ASSIGNREGISTER(dst, r_dst)
      ENDIF
      ASSIGNSOURCESOCKET(m, si)
      ASSIGNDESTINATIONSOCKET(m, si)
      ASSIGNMOVEBUS(m, mb)
      return TRUE
    ENDFOR
  ENDFOR
ENDFOR
return FALSE
```

no register is available, another RF is selected. The results of this straightforward application of RF partitioning are shown in Figure 9.13. Because the RFs, whose sockets are in the front of the list are picked relatively often, many RF-port conflicts occur. This resulted in a large performance penalty.

In an attempt to distribute the RF-port accesses more evenly across the RFs, we implemented a *best-fit* strategy. A register is selected that resides in the RF with the fewest accesses in instruction i where this new access will be scheduled. The RF sockets are sorted such that the sockets connected to the RFs with the fewest number of accesses in instruction i are in front of the list. The results of this strategy are also given in Figure 9.13. When using two partitions

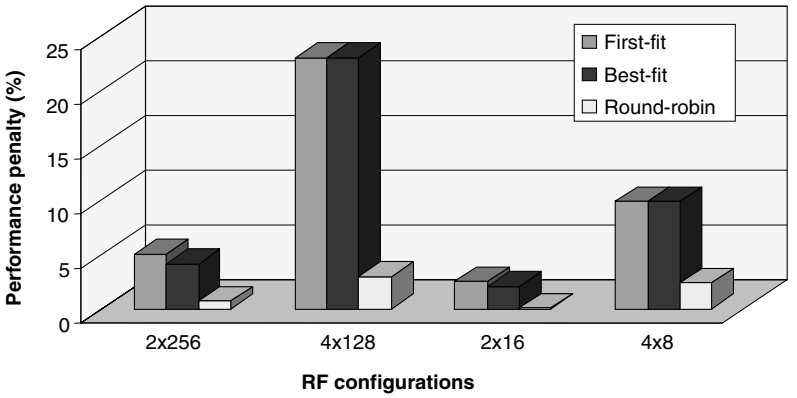


Figure 9.13: The results of the three local RF partitioning strategies: first-fit, best-fit and round robin.

the results improved somewhat. However, when using RFs with a single read port and a single write port, the best-fit strategy produces the exact same results as the first-fit approach. This is caused by the fact that in both strategies only those RFs can be selected that have no reference at all in the instruction in which the reference is scheduled. The orderings of the RFs with no reference are the same for both the best-fit and first-fit approach. This is also the weakness of this approach: the RF, which is in front of the original list, is selected relatively often, even when other RFs with no references are available. As a result, other, not yet scheduled references of the live range of the already assigned variable have a higher probability of RF conflicts with other references to the first RF.

A better strategy is to use another RF each time a register is required. This strategy is similar to the horizontal distribution as used in early assignment. We implemented a *round robin* strategy. This is accomplished by rotating the sockets in the socket list each time an attempt is made to assign a register to a variable. The results of the round robin strategy are shown in Figure 9.13. As can be seen, this method resulted in the smallest performance penalty.

9.4.2 A Global Heuristic

Like Ellis [Ell86] the local strategies only consider the RF-port usage in the instruction where the operation or transport that requires an RF-port is scheduled. However, RF-port assignments must be planned globally since a variable is accessed in other instructions too. Therefore, also all other accesses to a variable and all potential RF-port conflicts in their live range should be taken into consideration.

For example, consider the sequential code in Figure 9.14 and assume one RF-port for writing per RF. The transports defining the variables `v3` and `v5`

```
v1(r1) → add.o;  v2(r2) → add.t;
add.r → v3;
v4(r4) → sub.o;  #4 → sub.t;
sub.r → v5(r5);
```

Figure 9.14: Potential RF-port conflicts in sequential code.

```
v1 → add.o;  v2 → add.t;
add.r → v3;
v4 → sub.o;  #4 → sub.t;
sub.r → v5;
v8 → v9;
v6 → mul.o;  #10 → mul.t;
mul.r → v7;
v5 → st.o;   v7 → st.t;
```

a) Unscheduled TTA code.

r1	→ add.o	r2	→ add.t	
add.r	→ r2			
r3	→ sub.o	#4	→ sub.t	
sub.r	→ r3	r1	→ r4	

b) Partial schedule 1.

r1	→ add.o	r2	→ add.t		
add.r	→ r2	r1	→ mul.o	#10	→ mul.t
r3	→ sub.o	#4	→ sub.t	mul.r	→ r5
sub.r	→ r3	r1	→ r4		

c) Partial schedule 2.

r1	→ add.o	r2	→ add.t		
add.r	→ r2	r1	→ mul.o	#10	→ mul.t
r3	→ sub.o	#4	→ sub.t	mul.r	→ r5
sub.r	→ st.o	r1	→ r4	r5	→ st.t

d) Schedule.

Figure 9.15: Global RF-port conflicts.

both require an RF write port. Variable `v5` is already mapped onto `r5` and thus is already mapped onto an RF. In an efficient schedule, both accesses may be placed in the same instruction, and thus compete for an RF-port. When the result move of the addition is scheduled first, the register allocator should avoid that variable `v3` is mapped onto the same RF as variable `v5`.

Besides RF-port conflicts between not yet scheduled operations, also conflicts can arise between not yet scheduled and scheduled operations. Figure 9.15b shows the partial schedule, after scheduling of the first three operations of the sequential code of Figure 9.15a, under the assumption that the RFs only have one RF-port for reading. When scheduling the result of the multiply, a register and thus an RF, has to be selected; in this case register `r5` is selected (see Figure 9.15c). The selection of the RF has an impact on the scheduling of the trigger move of the store operation. This is shown in Figure 9.15d; when register `r5` is in the same RF as register `r1` an RF-port conflict arises between the read access of the copy and the read access of the trigger move of the store operation. Thus, a local decision has impact on a global scale.

Based on these observations it seems advantageous to avoid local as well as global RF-port conflicts. To accomplish this, an *RF-port conflict table* is build each time when integrated assignments seeks a register for a variable v. This table has as many entries as there are partitions. When scheduling transport

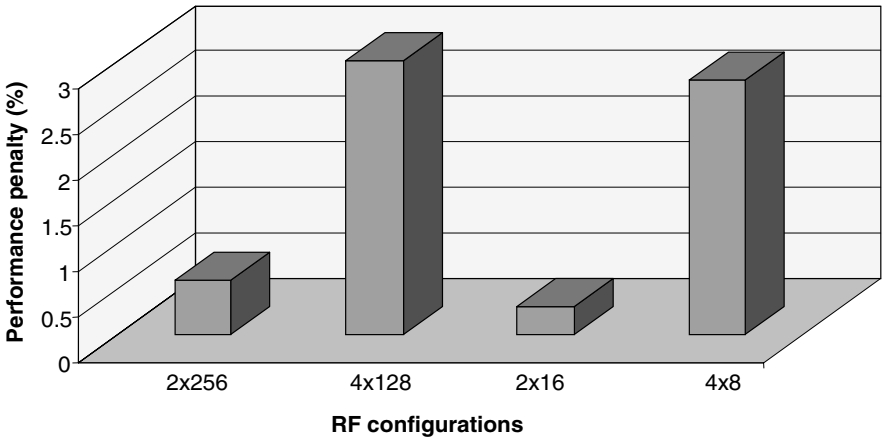


Figure 9.16: The results of the global RF partitioning strategy.

m referring to v, the code is scanned for potential RF-port conflicts. Potential conflicts are identified by inspecting the DDG for operations with already assigned registers and with the same access type (read or write)[4] that could be scheduled in the same instruction as m. For each potential conflict m_i, the table is updated with an RF-port conflict cost. The updated table entry is the entry that corresponds to the RF of the conflicting access m_i. Besides recording conflicts for the to be scheduled transport m, also potential RF-port conflicts with all other references to variable v are recorded by traversing the du-chain. It is decided to only consider the RF-port conflicts in the basic blocks containing the references. Including all RF-port conflicts is too time consuming; with region scheduling all operations that could be imported should also be considered each time an assignment is made. Furthermore, most of the potential RF-port conflicts with operations of other basic blocks will never occur in practice. In our experiments the RF-port conflict costs are equal to the expected execution frequency of transport m_i. The RFs are ordered with respect to their total RF-port conflict costs. The RF with the lowest total cost is selected first. The round-robin strategy is used as a fallback option, when two or more RFs have the same RF conflict cost.

The results of the global strategy are shown in Figure 9.16. They are close to that of the round-robin strategy. The general conclusion is that an ad-hoc approach such as round-robin is as good as a more sophisticated strategy. It should be noted that in both cases the performance loss caused by RF partitioning is small.

To our surprise, some of the results show an improvement when the RF is partitioned. For example the benchmarks *gzip* and *mulaw* show a small perfor-

[4]No shared read/write ports.

mance gain, see Appendix B. This effect can be explained by the same observation as was made in Section 7.6.3. When the region scheduler is successful in importing operations, a basic block can be removed (see Figure 7.11). However, the removal of this basic blocks may result in an overall negative effect. When using a partitioned RF, an RF-port conflict may hinder importing operations and thus will prevent that the basic block is removed.

9.5 Conclusions

In this chapter, the various effects of a single monolithic RF in the context of ILP processors are evaluated. Models are presented for the area, access time and power consumption. These models show that it is advantageous to partition the RF because the total area, access time and power consumption are reduced. However, partitioning may increase the number of executed cycles due to RF-port conflicts. We have shown that RF partitioning, when applied to TTAs, does not have to result in a large performance loss. Both, for early assignment and integrated assignment, the average performance loss can be kept below 5% with simple heuristics. We currently do not see a good method for using a partitioned RF in combination with late assignment. The performance results shown in this chapter are much better than the results reported for distributed RFs in [CDN93]; the reported performance losses for distributed RFs ranged from 4% to 40% for two partitions and from 15% to 75% for four partitions. The reason is twofold. First, partitioned RFs do not require extra copy operations between registers of different clusters. Second, in TTAs software bypassing and dead-result move elimination can be applied. This reduces the number of RF accesses and thus reduces the number of RF-port conflicts.

A partitioned RF requires less area. If we compensate for this, by adding extra registers to the partitioned RF, the performance losses caused by partitioning have been shown to disappear; the usage of a partitioned RF may even result in a small performance gain. If the register access time constrains the achievable cycle time of a processor, and is therefore critical in the overall performance of the system, then the performance loss due to the partitioning completely vanishes.

Finally, we summarize the advantages of using a partitioned RF versus a monolithic RF: (1) a partitioned RF is easier to realize due to the low complexity of the components, (2) the approach is scalable: the RFs can be duplicated to fit the architecture requirements, (3) the design can be made with available low cost SRAM cells, (4) the required chip area may reduce, (5) the access time of an RF with fewer ports is smaller and (6) the power dissipation reduces. Since the trend is towards the exploitation of more and more ILP, it is likely that the RF becomes a larger bottleneck. Our approach offers good performance as register (bandwidth) requirements double and redouble from current needs.

Summary and Future Research 10

In this dissertation we considered and resolved issues associated with register assignment, instruction scheduling, and partitioned RFs. Our overall investigation and achievements are summarized in Section 10.1. The contributions that stem from the described work are listed in Section 10.2. Section 10.3 discusses future research directions in the area of register assignment, ILP exploitation, and TTA research.

10.1 Summary

Microprocessors that are more powerful and high quality compilers are required to keep up with the ever-increasing complexity of embedded applications. In addition, the power consumption should be kept to a minimum, because many new devices are battery powered. Microprocessor research at the Delft University of Technology resulted into the development of the Transport Triggered Architecture (TTA) [Cor98]. This new architecture promises to fulfill the above mentioned requirements.

Chapter 2 discussed the main characteristics of TTA based processors. TTAs combine flexibility, modularity, and scalability. They can be tuned for any application or application area. The application's complexity, combined with a reduced time-to-market, makes a software based approach to embedded systems particular appealing today. Therefore, a high-level language compiler environment, based on aggressive instruction-level parallel (ILP) compiler technology, is a prerequisite. The used compiler infrastructure was discussed in Chapter 3. The benchmark suite, used for the experiments, consists of two TTA processors and 30 benchmark applications, including SPECint95 and multimedia benchmarks. The characteristics of these benchmarks were presented in Chapter 4.

The compiler is divided into a front-end and a back-end. The front-end translates a high-level language into an intermediate format. The back-end uses this intermediate format to generate parallel code. The main phases of the back-end are register assignment and instruction scheduling. The register allocator assigns registers to variables, and the instruction scheduler parallelizes the code. Applying register assignment first, limits the scheduler's ability to reorder operations. Applying scheduling first, results in schedules that require more registers than available. The interaction between register assignment and instruction scheduling has its impact on the produced code; decisions made by one phase can have negative effects on the other.

In this dissertation, various register assignment and instruction scheduling phase ordering strategies are investigated. To the best of our knowledge, it is the first time that three alternative strategies, early, late and integrated assignment, are compared and investigated in so much detail. In Chapter 5, approaches from literature towards the interaction of register assignment and instruction scheduling are discussed. The low performance of a strictly early assignment approach originates from the false dependences it introduces; they hinder the generation of efficient schedules, because they increase the critical path. Most approaches that try to avoid the introduction of false dependences use the data dependence graph (DDG) to make the register allocator *dependence-conscious* [GWC88, Pin93, NP93, AEBK94]. Also in the TTA compiler back-end a dependence-conscious approach is implemented. The results show a significant improvement compared to strictly early assignment. This is the first indication that register assignment and instruction scheduling must co-operate to achieve high performance code. Because register assignment is applied before instruction scheduling, the avoidance of false dependences is based on educated guesses. Consequently, the avoided false dependences may turn out not to be critical, while the ones that could not be avoided increase the critical path. Furthermore, the TTA specific optimizations, such as software bypassing and dead-move result elimination, result in a lower register pressure after scheduling than before scheduling. Unfortunately, an early assignment approach cannot exploit this optimization. Both observations show that there is still room for improvement.

Another strategy is late assignment; instruction scheduling is performed before register assignment. This strategy has the advantage that the instruction scheduler is not hindered by false dependences and, the TTA specific optimizations are visible for the register allocator. This seems advantageous, however, the instruction scheduler, in its attempt to reorder instructions to maximize ILP, may lengthen the live ranges of values and thus increases the contention for registers. If not enough registers are provided by the target processor, the data is written to memory, introducing spill code, which itself also requires registers. The increase in ILP can be nullified by the amount of spill code. This problem is identified in the literature [BSBC95]. To reduce the impact of the inserted spill code, rescheduling is usually applied. To lower

the register pressure so-called *register-sensitive* scheduling algorithms are proposed [GWC88, BEH91a, MPSR95] that limit the scheduling aggressiveness. An estimate of the register requirements is used for switching between an instruction scheduler that maximizes ILP and a scheduling method that reduces the register pressure. A register-sensitive instruction scheduler is implemented into the TTA compiler back-end. The performance gain, in relation with strictly late assignment, was disappointing. The main reason was that the number of register pressure reducing operations in the ready set is limited or non-existing. Furthermore, it was shown that the insertion of spill code, when using late assignment, is problematic for TTAs. Instead of adding spill code into the already scheduled code, spill code was added to the original code. The resulting code is then scheduled again. Our experiments indicate that for TTAs dependence-conscious early register assignment outperforms late assignment.

To address the problems of early and late assignment, a new register assignment method is developed: *integrated assignment*. The register assignment is performed during instruction scheduling. A register is assigned to a variable as soon as an operation is scheduled that refers to this variable. In Chapter 6, this idea has been presented in the context of basic block scheduling. To record the availability of registers in instructions, so-called Register Resource Vectors (RRVs) are introduced. With the use of these RRVs, dataflow information, and the DDG, the set of available registers for a variable is computed. In order to be complete, also methods to insert spill code and state preserving code during scheduling are presented. The insertion of spill code results in new short live ranges that also require registers. However, it was this resource that caused the spilling. To solve this problem the TTA specific properties are exploited. The scheduler is changed in such a manner that it is guaranteed that these new short live ranges, when needed, are software bypassed and that dead-result move elimination eliminates these short live ranges from the scheduled code. The results of this new approach showed that improvements, compared to a dependence-conscious early register assignment approach, of up to 100%, with an average of 20%, can be obtained when registers are scarce. Of course, when there are plenty of registers this new method should (and does) give the same performance as the early and late assignment methods.

The amount of exploitable ILP in basic blocks is limited. To justify the duplication cost of FUs and data paths in ILP processors, the ILP between operations of different basic blocks should also be exploited. In Chapter 7, it is shown how integrated register assignment can be applied in extended basic block schedulers. In the experiments, a region scheduler was used. Since operations can be moved from one basic block to another, the exploitable ILP increases, as well as, the register pressure. During the experiments, it was discovered that limiting the aggressiveness of the instruction scheduler might result in a performance increase. In order to limit the aggressiveness of the instruction scheduler, new heuristics are proposed. The results showed that our integrated assignment method, in combination with a region scheduler, per-

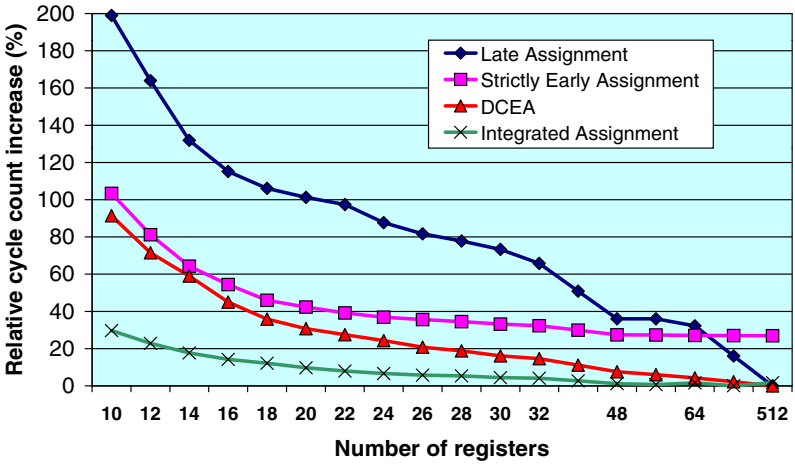


Figure 10.1: Relative cycle count increase of late assignment, strictly early assignment, dependence-conscious early assignment and integrated assignment for region scheduling and the TTA_{ideal} template.

forms very well. On average we obtained a 40% performance increase, compared to dependence-conscious early assignment, when registers are scarce. Individual benchmarks show performance gains up to 140%. The performance gain is even larger than for basic block scheduling.

To support our hypothesis that our approach is especially suitable for applications with a high register pressure, we integrated our method also with software pipelining. It is known that this instruction scheduling method increases the register pressure significantly, and produces high performance code. In Chapter 8 issues related to software pipelining in the context of integrated assignment are discussed. The results of the experiments showed that our conclusion was correct. The performance gain was even larger than for region scheduling. Normally, a high register pressure occurs in code with a large amount of exploitable ILP. This brings us to the conclusion that integrated assignment is especially suitable for high performance applications.

The performance improvement of integrated assignment over all other evaluated register assignment methods is given in Figure 10.1. This figure shows the cycle count increase for the TTA_{ideal} template of late assignment, strictly early assignment, dependence-conscious early assignment and integrated assignment relative to a TTA_{ideal} processor with 512 registers. Regions are used as the scheduling scope. The figure shows that the developed method is superior to all other methods. However, it must be noted that integrated assignment has a much higher engineering complexity than assignment methods based on graph coloring. This is mainly caused by the complexity of scheduling spill code.

Exploiting a high degree of ILP requires highly concurrent register access, yet it is difficult to provide fast, parallel, and global access to a single register file (RF). The RF represents a key component in the design of a fast data path since conflicting requirements in terms of short access time, large number of registers and large I/O data bandwidth, can drive the RF to quickly become a limiting factor for performance. Integrated register assignment requires fewer registers to achieve the same high quality code as early and late assignment. Consequently, the RF can be smaller; this results in a reduction in silicon area, power consumption and access time. In Chapter 9, it was observed that besides the number of registers, the number of RF-ports largely determines silicon area, power consumption and access time. It was also observed that a partitioned RF, with the same number of registers and RF-ports, is smaller, consumes less power and is faster than a single monolithic RF. Therefore, it was proposed to use several small RFs instead of one single monolithic RF. New methods are developed to make early and integrated assignment suitable for the partitioned RF. Because of the fewer ports per RF, the probability of RF-port congestion per RF increases. Heuristics are developed to minimize the impact of RF-port conflicts. Experiments showed that only a small loss in performance, due to access conflicts, has to be accepted when partitioning the RF. When the area saved by the partitioning is spend on extra registers even a performance increases can be observed. This increase is even larger when the reduced access time is taken into account.

10.2 Contributions

The major contributions of this thesis are summarized below.

- For the first time, mathematical formulations are developed for various dependence-conscious early assignment methods found in literature. In addition, the relations between these approaches are mathematically formulated.
- A late assignment method in the context of TTAs is developed. The problems related to the insertion of operations in already scheduled TTA code are addressed.
- A new approach towards the integration of register assignment and basic block scheduling is developed [JC97b]. It is shown that the introduced algorithm can gracefully handle the insertion of spill code and state preserving code. New heuristics are proposed and evaluated. It is demonstrated that this method outperforms all other evaluated phase ordering strategies.
- The developed integrated assignment method is extended to region scheduling in order to exploit a larger amount of ILP [JC98, CJA00].

Heuristics are developed to limit the aggressiveness of the instruction scheduler in order to increase the performance. It is shown that this method outperforms all other evaluated phase ordering strategies. It is very effective in exploiting ILP when the register pressure is high.

- In order to demonstrate the effectiveness of integrated assignment, it is applied in combination with software pipelining [JCK98], a very aggressive instruction scheduling method. The experiments showed that the developed method is indeed very suitable for code with a large register pressure.
- Insight has been gained in the delay, area and power consumption problems of multi-ported register files. New RF configurations are proposed to overcome these problems. In addition, new methods have been developed to minimize the performance penalty, when early assignment is applied to TTA processors with a partitioned RF [JC95].
- It is shown that the concept of partitioned RFs can easily be added to integrated assignments. Experiments show that the performance penalty due to RF-conflicts can be limited.

Another contribution related to the research presented in this dissertation is Controlled Node Splitting, a new technique to restructure the control flow graph in order to facilitate the exploitation of ILP. This work resulted in a conference and a journal paper [JC96, JC97a].

10.3 Proposed Research Directions

The work described in this dissertation can, in our opinion, be continued in the following directions.

Early Assignment Improvements

Because of the low complexity of graph coloring in contrast to integrated assignment it is interesting to investigated how early assignment can be further improved. Based on our experience with early assignment algorithms and their results, the following possible improvements were identified.

1. **Better interference graphs.** During the analysis of the various dependence-conscious early assignment strategies, it was observed that none of the parallel interference graphs makes a clear distinction between *real* interference edges and *false* dependence prevention edges. With real interference edges, the interferences that always will be present, independent of the generated schedule, are meant. These edges are denoted with E_{min}. With false dependence prevention edges we refer to the interferences that might be present in a schedule and is denoted as $E_{false} = E_{fdp} - E_{min}$. The new interference graph with a

clear cut between both type of edges can now be constructed as: $G_{dc} = (N_{var}, E_{min}, E_{false})$. Note that this approach is a combination of the work of Ambrosch [AEBK94] and Pinter [Pin93], see Section 5.1. Future research may lead to heuristics that use both type of interference edges to make a better trade-off between spilling and the addition of false dependences.

2. **Parallel interference graph pruning.** The parallel interference graph tends to have many interference edges, especially when the scheduling scope is extended beyond basic block boundaries. Methods are developed to prune the graph. For example, to only include forward false dependences, see Section 5.1.3. This may however, sometimes lead to an unnecessary performance loss. Other pruning methods can be developed. One can think of including more scheduling decisions, like the impossibility to execute operations from various basic blocks in parallel, due to a restriction in speculative execution. Because not all edges are important, methods should be developed to identify only the important ones. For example, a pre-scheduler could help to identify false dependences that should be avoided.
3. **False dependence-conscious operation reordering.** The direction of a false dependence added by register assignment depends on the operation order in the sequential code. In order to reduce the impact on the critical path, one could research the impact of a false dependence-conscious operation reorder technique.
4. **False dependence elimination during scheduling.** In an early assignment approach, software bypassing and dead-result move elimination, can eliminate false dependences. For example, consider Figure 6.19c. The false dependence that hindered the parallel execution of the operations is hidden by software bypassing and dead-result move elimination because the live ranges are not visible anymore in the resulting code. When both the addition and the store are scheduled, the false dependence can be removed from the DDG. This allows an early assignment scheduler to generate parallel code as shown in Figure 6.19d.

General Register Assignment Issues

Besides improvements in early assignment, also other register assignment topics require further research. These topics are:

1. **Live range splitting.** All discussed register assignment methods spill the complete live range of a variable. When a live range consists of many definitions and uses, one could decide only to spill part of the live range. Hence, the variable resides for some time in memory and the remaining time in a register. Because fewer reloads are necessary, a performance

increase is expected. The main problem to be solved is: where in the code to switch between a register or memory location.

2. **Development of better heuristics.** The presented experiments indicate that with the use of heuristics improvements can be obtained. However, some of the heuristics do not have the same impact on all benchmarks (for example the GSC_5 heuristic). Analysis of these benchmarks may result in a top-level heuristic that decides which heuristic or which combination of heuristics to use. This heuristic can be different for different parts of the code. Another research direction can be the development of a framework that automatically evaluates various heuristics and based on the results selects the best heuristic. This framework can, for example, be used to decide whether to include in the total schedule, a loop's software pipelined code or it's region scheduled code.

3. **Predicate analysis.** The current register assignment implementations ignore predicate or guarded execution analysis to reduce the register pressure. The major reason to ignore this optimization was that guarding is not used extensively. However, ILP enhancing techniques exist which heavily depend on predicates. In order to fully benefit from these techniques, research should be conducted to allow non-interfering variables that overlap in time to share a common register. A good starting point for this research is [ED95a].

4. **Basic block insertion.** The used region scheduler removes a basic block when all operations of it are imported to other basic blocks. However, as discussed in Section 7.6.3, this is not always advantageous, especially when in an if-then-else construction the outcome of the jump is strongly biased towards one branch, and the basic block in the other branch is removed. As a result, operations below the join point of the two branches cannot be imported in the first branch, because this will result in a violation of the single copy on a path rule. Research should be conducted to decide when to remove basic blocks, or when to add basic blocks in order to increase the exploitable ILP.

5. **Creating extra temporary registers.** Besides the registers in the RF, also the registers in the FUs can be used for storing variables. For example, one could decide to perform an addition with zero to create a place holder for a variable with a short live range.

6. **Single ported RFs.** In this thesis, the research was limited to RFs with separate read and write ports. Combining these two functions in one RF-port may reduce the cost. The TTA compiler can already handle RFs with multiple RF-ports that combine reading and writing. However, how to compile for an RF with a single RF-port is still an open issue. This is especially a problem for copy operations, because copies require a read and a write in the same cycle. A possible solution is to replace the copy

```
#100 → mul.t;   v3 → mul.o
mul.r → v1;
#17 → add.o;    v1 → add.t;
add.r → v2;
#200 → st.t;    v1 → st.o;
```

a) Code fragment.

```
#100 → mul.t;   v3 → mul.o
mul.r → add.t;  #17 → add.o;  mul.r → st.o;
#200 → st.t;    add.r → v2;
```

b) Resulting schedule.

Figure 10.2: Optimistic scheduling.

with an addition that adds zero to the operand of the copy.

7. **Duplicated variables.** When using RF partitioning and performance is deteriorated by RF-port conflicts, it could be an option to store a variable in multiple RFs [BS91]. This increases the probability that the value of a variable can be read from an RF in a certain cycle and thus reduces the number of RF-port conflicts. Care must be taken to ensure that all duplicates of the variable are consistent with each other.

8. **FUs with RF run-outs.** The idea of run-outs [Cor98] has the advantage that results may stay longer in FUs. Consequently, more values can be software bypassed. This may result in a reduction of the required number of RF-ports and registers.

9. **Optimistically assuming software bypassing and dead-result move elimination.** Even integrated assignment may superfluously spill variables. For example, consider the code fragment in Figure 10.2a. Assume the scheduler selects the operations for scheduling in the order `mul`, `add`, `st`. This sequence requires two registers because the variables `v1` and `v2` are simultaneously live, when the result of the addition is scheduled. When only one register is available, spill code is inserted. However, as shown in Figure 10.2b the total schedule only requires a single register. The problem is that only after scheduling of the store, it becomes clear that only a single register is required. A solution to this problem is to optimistically assume that software bypassing will free a register and thus the scheduler continues scheduling. When after a number of scheduling steps it becomes evident that no registers become available, backtracking must be used to generate correct code.

10. **Alternative phase orderings.** Besides early, late and integrated assignment strategies, one can also research alternative phase orderings. For

example, one could research a phase ordering, in which first the most frequently executed region is scheduled, after which registers are assigned to the variables referenced in this region. Because some variables are removed from the code due to software bypassing and dead-result move elimination, more register might be available for variables that are live across this region, but are never referenced in it. The same method can be applied to the other regions as well. Other approaches from literature are known that divide the registers set in local and global registers [BEH91a, HA99]. Local registers are assigned to variables that are only referenced in scheduling scopes without loops (basic blocks or trees), while the global registers are assigned to variables that are live across the boundaries of the used scheduling scope. For the assignment of both register sets a different register assignment strategy can be used. In [Mes01] an interesting approach is presented in the context of constrained based scheduling. Analysis and assignment steps are iteratively applied to prune the scheduling search space. Eventually, a schedule is generated, if feasible, that complies with the given constraints.

General TTA related topics

There are many other topics related to the research presented in this dissertation. Some major research areas for future research are listed below:

1. **Clustered architectures.** Advances in silicon technology allow the increase of the number of FUs and data paths in order to increase performance. However, the enormous amount of connections to the move buses and length of these buses, may result in a large cycle time and hence the execution time may increase significantly. This problem can be tackled with the use of a clustered architecture. Figure 10.3 shows a TTA processor with four clusters. Because the number of connections to a single bus and its length are reduced, the cycle time is reduced in comparison to a single cluster architecture. Communication between clusters is achieved by inter-cluster RFs. Although clustered architectures reduce the cycle time, it is hard to exploit the increased exploitable ILP. Therefore, new compiler technologies have to be developed to exploit ILP in the context of clustered TTAs [RCL01].
2. **Enhancing exploitable ILP.** Although the methods presented in this dissertation already exploit a reasonable amount of ILP, methods exists that even may increase the exploited ILP. For example, if-conversion can be applied more aggressively. Other examples are source code transformations for loops. In addition, research should also be devoted in developing new ILP enhancing techniques.
3. **Compilation for caches and local memories.** Research should be directed towards compilation techniques for caches and local memories.

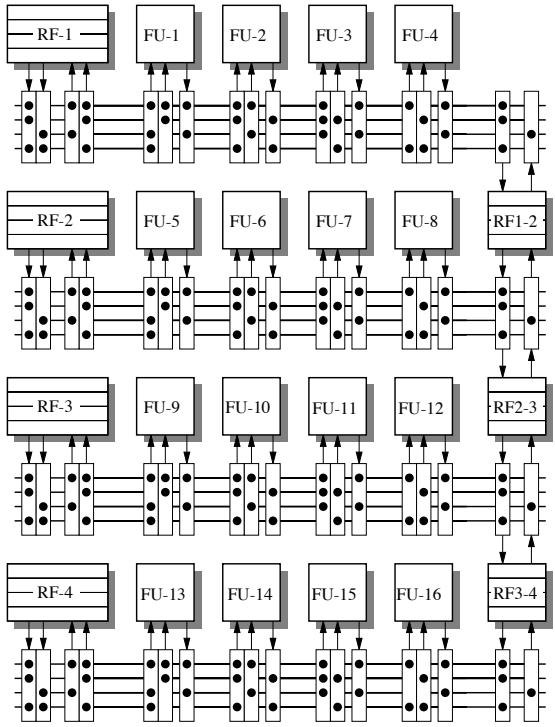


Figure 10.3: Example of a clustered TTA.

Caches are a very suitable technique to increase performance. This performance increase can even be further improved when the compiler is made cache aware. In addition, many embedded applications benefit from multiple local memories. To make use of these memories new compiler techniques must be developed.

4. **Code density.** A drawback of VLIWs and TTAs is their code size. Empty slots in an instruction are filled with no-ops. In embedded systems, code density is an important issue. Therefore, techniques have to be developed and evaluated that solve this problem.

5. **Reduced RF-FU connectivity.** The current implementation of the TTA back-end requires that there is always a path between any FU-port and the RF(s). This constraint originates from the early assignment strategy because the register allocator is unaware of the connectivity between RFs and FUs, and on which FU an operation will be scheduled. When using integrated assignment this restriction can be released. When no direct path exists between the RF that holds the required value and the FU that requires it, extra copy operations can be inserted during scheduling.

6. **Programming tools.** The amount of exploitable ILP not only depends on the compiler, but also on the programmer. Tools should be developed that inform the programmer how the application should be improved at source code level.

Integrated Assignment Benchmark Results

A

In the Sections 6.6.4, 7.6.3 and 8.4.2 the average performance gains of integrated assignment compared to dependence-conscious early register assignment are given. In this appendix the individual benchmark results of these experiments are listed:

- Table A.1: Integrated assignment versus dependence-conscious early assignment, using basic block scheduling and the TTA$_{ideal}$ template.
- Table A.2: Integrated assignment versus dependence-conscious early assignment, using basic block scheduling and the TTA$_{realistic}$ template.
- Table A.3: Integrated assignment versus dependence-conscious early assignment, using region scheduling and the TTA$_{ideal}$ template.
- Table A.4: Integrated assignment versus dependence-conscious early assignment, using region scheduling and the TTA$_{realistic}$ template.
- Table A.5: Integrated assignment versus dependence-conscious early assignment, using software pipelining and the TTA$_{ideal}$ template.
- Table A.6: Integrated assignment versus dependence-conscious early assignment, using software pipelining and the TTA$_{realistic}$ template.

Table A.1: Speedup in percentages of integrated assignment compared to dependence-conscious early assignment, using basic block scheduling and the TTA$_{ideal}$ template.

Benchmark	Number of Registers														
	512	64	48	32	30	28	26	24	22	20	18	16	14	12	10
a68	0.0	0.0	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	0.0	0.0	0.0	-0.2	0.1	0.8	1.9
bison	0.0	0.0	0.0	-0.1	-0.1	0.0	0.0	-0.1	0.1	0.3	0.6	0.5	0.7	1.6	1.9
cpp	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.1	0.5	1.3	1.5	2.7	6.2
crypt	0.0	0.0	0.0	6.2	16.9	18.1	8.4	19.7	17.8	21.8	37.5	73.2	91.2	85.4	89.7
diff	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.7	0.8	0.9	2.1	3.5	4.7
expand	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
flex	0.0	0.0	-0.1	-0.1	-0.1	0.0	0.1	0.3	0.9	1.1	1.8	2.1	3.3	4.9	6.8
gawk	0.4	0.4	0.4	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.0	0.2	-0.1	-0.3	0.6
gzip	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.6
od	0.0	0.0	-0.1	-0.1	-0.2	-0.1	0.0	-0.2	1.0	1.5	3.0	4.1	3.9	3.4	5.3
sed	0.1	0.9	-2.2	-0.6	0.0	0.9	1.4	1.4	1.0	1.5	2.0	3.0	3.6	5.6	7.7
sort	0.1	0.1	0.1	0.6	1.0	1.0	2.1	2.6	6.3	7.2	9.2	16.4	25.5	35.9	40.5
uniq	0.1	0.1	0.1	-0.1	-0.1	-0.2	-0.3	0.3	0.4	0.4	0.0	3.0	5.0	3.9	12.7
virtex	0.1	0.1	0.1	0.1	0.1	0.0	0.0	0.4	0.5	0.5	0.9	1.5	2.0	3.7	6.5
wc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
099.go	0.0	0.0	-0.1	-0.1	0.0	-0.1	0.0	-0.5	-0.5	-0.5	-0.3	0.6	1.3	1.1	1.0
124.m88ksim	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	-0.1	0.1	0.5	5.8
129.compress	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.6	4.0	4.4	4.0
132.ijpeg	0.0	0.1	0.3	6.4	8.2	9.0	13.0	14.6	21.2	28.7	35.9	52.1	71.4	83.8	105.5
147.vortex	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.2	0.6	0.7	0.7	1.2
instf	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.1	5.1	6.9	6.3	7.5	11.9	11.5
mulaw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
radproc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	4.4	8.3	7.1	2.4	7.4	20.9	30.2
rfast	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	3.5	15.7	21.1	18.7	22.4	35.2	33.9
rtpse	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.6	3.9	10.1	21.7	18.5	19.2	28.0	28.1
djpeg	0.0	0.3	0.7	11.1	14.7	17.9	21.9	24.2	29.2	35.7	42.9	42.8	45.9	86.7	116.1
rawcaudio	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-1.2	40.3
mpeg2decode	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.3	1.8
mpeg2encode	0.0	0.0	0.0	18.0	20.2	23.6	27.0	30.4	31.7	32.5	34.2	35.8	37.9	40.7	46.6
unepic	0.0	0.0	0.0	0.1	0.2	2.6	2.6	2.0	2.0	5.9	6.9	15.6	21.5	27.7	27.4
average	0.0	0.1	0.0	1.4	2.0	2.4	2.6	3.3	4.2	5.9	7.8	10.1	12.6	16.4	21.3

Table A.2: Speedup in percentages of integrated assignment compared to dependence-conscious early assignment, using basic block scheduling and the TTA$_{realistic}$ template.

Benchmark	Number of Registers														
	512	64	48	32	30	28	26	24	22	20	18	16	14	12	10
a68	-0.1	-0.1	-0.2	-0.3	-0.3	-0.3	-0.3	-0.3	-0.2	-0.2	-0.2	-0.3	0.1	0.8	1.9
bison	-0.1	-0.1	-0.1	-0.1	-0.2	-0.2	-0.2	-0.3	-0.1	0.2	0.4	0.2	0.5	1.4	1.7
cpp	-0.1	-0.1	-0.1	-0.1	0.0	-0.1	0.0	-0.1	0.0	0.1	0.6	1.3	1.3	2.8	6.0
crypt	-0.1	3.4	4.6	6.8	12.7	9.8	9.4	8.0	12.0	13.0	3.1	32.3	37.5	50.3	42.8
diff	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.7	0.8	0.9	2.1	3.6	4.7
expand	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
flex	0.0	0.0	-0.1	-0.2	-0.3	-0.3	-0.3	-0.1	0.4	0.6	1.4	1.5	2.5	4.1	6.0
gawk	0.5	0.4	0.5	0.8	0.6	0.6	0.5	0.4	0.3	0.3	0.0	0.3	-0.2	-0.7	0.2
gzip	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.4
od	0.0	0.0	-0.2	-0.3	-0.5	-0.4	-0.4	-0.8	0.2	0.7	2.3	3.0	3.2	2.8	4.7
sed	-0.1	-0.2	-3.4	-2.9	-2.2	-1.3	-0.9	-1.0	-1.2	-0.7	-0.4	0.3	0.6	2.5	4.0
sort	0.0	0.1	0.3	0.7	1.5	1.5	2.3	2.8	6.5	6.1	8.1	14.6	21.4	29.4	31.8
uniq	0.1	0.1	0.1	-0.3	0.1	-0.1	0.0	0.4	0.5	0.5	0.0	3.0	4.3	3.2	11.7
virtex	0.0	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.1	0.2	0.5	1.0	1.7	3.4	5.9
wc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
099.go	0.0	-0.1	-0.3	-0.7	-0.8	-0.9	-0.9	-1.4	-1.4	-1.5	-1.7	-0.9	-0.6	-0.1	0.7
124.m88ksim	0.3	0.3	0.3	0.3	0.3	0.3	0.1	0.2	0.1	0.2	0.0	-0.1	-0.4	0.3	5.0
129.compress	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.5	4.2	2.6	2.3
132.ijpeg	-0.3	0.5	-0.2	4.0	4.1	3.9	5.2	6.6	9.9	14.4	15.1	28.5	41.5	40.9	55.9
147.vortex	0.0	0.0	-0.1	-0.1	-0.1	-0.1	-0.2	-0.3	-0.5	-0.6	-0.4	0.4	0.5	0.6	1.1
instf	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.1	4.6	5.8	7.0	9.5	10.0	8.5
mulaw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
radproc	0.0	0.0	0.0	-0.1	-0.1	-0.2	-0.2	0.4	3.4	5.2	5.3	4.7	4.7	22.2	21.2
rfast	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	3.1	13.0	15.9	19.1	25.5	26.8	21.5
rtpse	0.1	0.1	0.1	0.0	0.1	0.0	0.0	0.5	3.4	8.1	17.5	19.8	21.2	22.2	20.2
djpeg	-0.1	-0.2	-0.2	5.9	9.8	10.0	13.2	15.8	18.4	20.8	22.1	19.7	22.6	52.0	55.9
rawcaudio	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-1.2	40.3
mpeg2decode	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.2	1.8
mpeg2encode	0.0	-0.3	0.1	8.2	10.0	12.6	14.6	17.5	18.5	19.3	20.0	21.5	23.2	25.9	28.2
unepic	-0.3	-0.3	-0.3	-0.2	-0.1	0.6	0.6	-2.3	-4.0	-1.5	-2.7	3.7	8.8	13.6	12.0
average	0.0	0.1	0.0	0.7	1.2	1.2	1.4	1.6	2.4	3.4	3.8	6.1	7.9	10.7	13.2

Table A.3: Speedup in percentages of integrated assignment compared to dependence-conscious early assignment, using region scheduling and the TTA$_{ideal}$ template.

Benchmark	Number of Registers														
	512	64	48	32	30	28	26	24	22	20	18	16	14	12	10
a68	1.5	1.5	1.5	1.5	1.4	1.5	1.9	2.0	1.9	1.6	2.2	2.9	2.9	2.5	6.9
bison	2.0	2.2	2.0	1.8	2.0	2.5	2.7	3.3	2.4	4.2	5.1	5.6	6.1	8.7	8.5
cpp	2.4	2.3	2.2	1.6	1.5	1.9	2.0	1.8	1.5	1.5	-1.8	0.7	2.3	4.4	14.1
crypt	0.2	1.3	1.9	4.7	10.0	13.9	21.2	31.1	29.3	14.8	32.3	54.8	56.5	89.9	94.7
diff	0.5	0.5	1.2	13.0	13.0	12.8	11.6	12.5	13.9	13.8	13.6	15.3	16.4	16.4	19.5
expand	-2.7	-1.6	0.5	-0.3	-0.5	0.1	1.0	-1.2	0.7	-9.7	-4.3	-4.5	-3.6	-3.3	-9.1
flex	1.4	1.4	1.5	1.4	1.4	0.4	0.7	2.5	2.3	6.0	3.6	6.6	8.0	11.1	14.5
gawk	1.9	1.9	1.2	0.6	0.7	1.0	1.1	0.9	0.8	2.1	2.5	2.5	1.6	0.6	2.1
gzip	3.3	3.4	3.6	2.2	2.9	2.9	3.5	6.0	5.9	5.9	5.9	6.9	6.0	6.7	27.9
od	1.0	1.5	1.4	2.6	-0.1	-1.1	2.0	3.7	5.2	14.3	9.9	16.3	30.3	32.1	30.9
sed	1.6	4.5	5.4	10.1	11.9	13.5	13.5	12.4	13.9	14.3	14.9	18.1	15.6	18.7	21.3
sort	1.8	0.6	1.4	1.0	-0.7	1.2	5.9	12.5	15.3	23.8	28.7	59.7	55.7	69.1	81.4
uniq	0.3	0.0	1.1	3.1	2.3	3.1	9.4	10.9	10.0	9.1	12.3	10.6	14.4	17.3	23.3
virtex	2.5	2.5	2.3	2.7	3.3	3.3	3.5	4.6	6.6	7.1	8.6	10.7	11.8	14.1	15.6
wc	0.0	0.0	0.0	9.2	9.2	9.2	9.2	9.2	9.2	8.7	8.2	7.9	7.4	-11.9	-6.5
099.go	2.0	2.0	2.2	2.4	2.5	2.8	3.7	5.7	5.7	6.1	7.6	8.6	9.0	10.3	8.4
124.m88ksim	1.6	1.7	1.8	3.0	2.9	2.9	2.6	1.8	2.0	2.6	2.6	2.5	2.8	4.6	14.8
129.compress	0.0	0.0	0.0	4.4	4.4	4.4	6.1	6.1	6.1	6.1	8.8	28.9	29.3	29.9	31.4
132.ijpeg	6.8	13.7	18.7	34.0	44.5	49.8	51.1	65.1	72.9	78.7	84.1	104.9	96.1	131.5	140.0
147.vortex	0.4	0.4	0.2	0.7	0.6	1.0	0.8	1.5	1.3	1.6	2.8	4.1	4.7	4.9	6.0
instf	0.9	1.8	2.1	5.5	7.9	10.8	9.9	14.6	20.7	22.8	25.7	35.7	65.3	73.4	79.5
mulaw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
radproc	0.3	1.4	2.1	1.6	2.0	2.0	5.9	7.7	17.4	19.4	17.9	25.2	44.6	39.0	57.8
rfast	2.5	3.1	3.9	5.4	3.7	9.1	15.5	24.8	24.2	27.5	36.0	38.0	63.8	80.2	102.6
rtpse	1.1	2.9	3.1	3.2	3.0	2.7	2.7	3.3	5.0	15.8	18.4	31.3	38.9	43.9	66.0
djpeg	6.7	19.2	28.3	53.7	61.8	67.7	64.3	69.7	80.0	98.6	90.7	92.0	113.8	144.6	128.0
rawcaudio	0.5	0.5	1.0	1.0	2.6	2.6	2.6	2.6	6.0	6.0	5.5	1.1	80.2	44.3	57.5
mpeg2decode	1.1	1.2	1.2	1.4	1.6	1.8	1.5	1.8	1.8	2.3	2.1	2.3	5.6	7.1	42.2
mpeg2encode	8.3	49.8	83.9	77.8	86.4	84.8	89.6	86.6	82.4	70.7	71.7	78.3	83.3	84.5	88.8
unepic	0.2	0.4	1.1	0.1	1.3	9.8	11.4	16.6	12.9	13.7	15.8	23.1	31.1	42.8	46.9
average	1.7	4.0	5.9	8.3	9.4	10.6	11.9	14.0	15.2	16.3	17.7	23.0	30.0	33.9	40.5

Table A.4: Speedup in percentages of integrated assignment compared to dependence-conscious early assignment, using region scheduling and the TTA$_{realistic}$ template.

Benchmark	Number of Registers														
	512	64	48	32	30	28	26	24	22	20	18	16	14	12	10
a68	0.7	0.7	0.7	1.1	1.2	1.1	1.7	1.6	1.8	1.4	0.9	1.3	2.2	2.3	3.3
bison	1.5	1.8	1.5	1.5	1.7	1.7	2.4	2.8	3.1	4.0	4.0	4.4	5.5	8.1	4.0
cpp	1.0	1.2	1.2	0.9	1.1	0.5	0.2	0.3	-0.5	0.6	2.2	2.5	3.0	5.7	11.7
crypt	0.0	0.8	1.6	5.7	8.2	9.6	4.5	6.5	-2.8	12.5	17.8	5.1	8.3	29.9	29.1
diff	0.4	0.4	1.1	11.1	10.1	9.9	9.9	11.5	12.8	12.6	12.3	12.8	13.3	13.3	18.1
expand	0.7	2.3	2.8	0.9	3.4	4.5	2.9	2.7	3.2	1.9	-9.1	-6.7	-8.8	-8.0	-11.9
flex	1.6	1.6	1.8	0.3	-0.3	0.8	0.4	2.4	2.1	4.6	2.1	5.3	6.6	10.1	12.1
gawk	1.9	1.9	1.8	1.9	1.9	1.9	1.9	1.8	1.4	2.9	3.4	3.4	2.1	0.9	2.2
gzip	4.0	4.1	4.5	-0.7	1.5	3.4	3.3	6.0	5.8	5.4	5.2	6.5	6.8	3.8	25.2
od	0.7	1.2	0.8	1.2	-0.8	-1.5	1.7	2.6	2.4	10.1	6.2	9.5	21.0	24.0	23.3
sed	-0.5	2.0	0.6	4.2	5.1	6.2	6.9	5.9	6.4	5.8	6.4	8.3	6.6	6.7	10.6
sort	0.4	2.6	3.0	2.7	2.9	1.9	1.9	7.2	7.3	15.1	19.6	45.2	42.2	51.8	57.7
uniq	-0.8	1.6	0.1	0.7	0.7	1.9	7.3	8.7	8.0	7.8	10.4	11.4	11.1	12.1	21.5
virtex	1.6	1.1	1.3	1.6	1.9	1.8	1.5	2.2	4.1	4.1	4.5	6.6	6.1	9.8	11.0
wc	0.0	0.0	0.0	11.9	11.9	11.9	11.9	11.9	11.9	11.9	9.3	11.1	12.8	5.4	10.3
099.go	1.3	1.2	1.3	1.2	1.2	1.1	1.5	3.0	3.3	2.1	2.2	3.3	5.0	6.5	4.4
124.m88ksim	0.5	0.5	0.4	1.0	0.9	1.3	-0.3	-0.2	1.3	1.7	2.0	2.0	2.1	4.1	13.6
129.compress	0.1	0.1	0.0	4.2	4.3	4.2	5.9	5.9	5.9	6.6	8.8	27.2	28.0	29.0	30.8
132.ijpeg	0.8	0.5	2.5	14.5	18.1	17.8	18.7	21.5	24.3	25.0	28.0	30.7	30.1	36.3	35.5
147.vortex	0.4	0.6	0.8	0.5	0.4	0.8	0.8	0.8	1.1	1.4	1.9	2.9	3.8	4.0	-1.2
instf	0.0	0.1	-2.5	-3.2	-1.1	-1.7	-1.7	-0.6	2.6	4.8	10.6	25.1	43.2	40.2	45.9
mulaw	0.0	0.0	0.0	2.4	2.4	2.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	2.9
radproc	-0.1	-0.6	-0.8	-0.5	-0.4	0.1	2.3	4.6	11.5	9.0	10.7	22.9	30.4	30.1	38.6
rfast	-0.1	-0.3	-3.1	-4.2	-1.9	-2.4	-0.2	0.6	1.5	3.2	13.5	29.6	48.9	40.1	57.1
rtpse	0.2	0.0	-0.1	0.2	0.1	-0.1	0.1	2.3	5.5	10.8	12.1	27.3	26.2	29.2	41.4
djpeg	0.3	-1.7	4.6	17.8	20.9	27.8	26.0	30.1	39.0	43.9	41.7	43.0	62.6	78.3	54.0
rawcaudio	0.7	0.7	0.7	0.8	4.7	4.7	4.8	4.8	6.0	5.9	3.7	-2.6	68.4	37.0	56.7
mpeg2decode	0.1	0.1	0.1	-1.3	-1.2	-1.1	-1.3	-1.0	-1.0	-0.6	-0.7	0.9	5.2	4.7	32.2
mpeg2encode	0.5	9.1	43.3	51.3	51.2	53.9	49.9	51.1	48.9	46.4	46.9	45.7	49.6	48.7	54.8
unepic	-0.3	-0.3	0.2	-1.6	-1.0	4.9	4.4	7.4	4.6	4.8	5.6	10.7	10.7	16.9	17.9
average	0.6	1.1	2.3	4.3	5.0	5.7	5.8	6.9	7.5	9.0	9.5	13.3	18.6	19.5	23.8

Table A.5: Speedup in percentages of integrated assignment compared to dependence-conscious early assignment, using software pipelining and the TTA$_{ideal}$ template.

Benchmark	Number of Registers														
	512	64	48	32	30	28	26	24	22	20	18	16	14	12	10
a68	1.4	1.5	1.5	1.5	1.4	1.6	1.9	2.0	2.4	2.2	2.5	2.9	2.9	2.7	7.0
bison	1.8	2.0	1.8	1.9	2.1	2.6	2.6	3.2	2.5	4.0	5.8	5.6	5.9	6.6	7.0
cpp	2.4	2.3	2.2	1.6	1.5	1.9	1.9	1.7	1.4	1.5	-2.0	0.6	2.3	4.3	13.6
crypt	0.2	0.2	0.2	8.8	23.6	27.7	13.8	25.4	40.9	39.3	47.1	88.1	142.7	121.1	119.1
diff	0.4	0.4	1.2	12.9	12.9	12.7	11.5	12.8	16.8	19.5	19.5	19.0	23.1	28.5	39.7
expand	-2.7	-1.6	0.5	-0.3	-0.5	0.1	1.0	-1.2	0.7	-9.7	-4.3	-4.5	-3.6	-3.3	-9.1
flex	1.4	1.5	1.6	1.1	0.9	0.5	0.7	0.7	1.1	3.9	3.0	5.0	7.3	8.5	12.0
gawk	1.9	1.9	1.2	0.6	0.7	1.0	1.1	0.9	0.8	2.1	2.5	2.5	1.6	0.6	2.1
gzip	6.3	6.4	6.6	2.4	2.8	3.3	3.7	6.5	6.3	6.5	12.0	12.5	11.8	21.4	41.6
od	0.6	1.2	0.9	2.2	0.4	-0.4	1.6	3.8	4.3	7.3	6.5	9.2	12.4	10.2	21.2
sed	1.6	4.3	5.4	10.0	11.8	13.1	13.1	12.0	13.4	14.4	15.2	17.3	14.8	18.2	20.6
sort	1.9	0.7	1.3	0.5	-1.6	0.7	7.7	10.3	14.7	24.6	30.7	55.0	57.4	77.4	92.5
uniq	0.3	-0.3	1.1	2.4	2.4	2.4	8.9	9.6	9.4	8.9	11.7	11.3	13.5	16.7	21.6
virtex	2.6	2.6	2.4	2.7	3.4	3.8	4.1	5.0	6.6	6.8	9.1	10.3	13.6	15.0	15.8
wc	0.0	0.0	0.0	9.2	9.2	9.2	9.2	9.2	9.2	8.7	8.2	7.9	7.4	-11.9	-6.5
099.go	2.0	2.0	2.2	2.3	2.5	2.7	3.7	5.6	5.7	5.8	7.6	7.6	9.0	9.8	7.2
124.m88ksim	4.5	4.6	4.7	4.9	4.6	4.5	3.4	3.9	3.6	4.1	3.7	3.2	4.3	-0.2	-1.7
129.compress	0.0	0.0	0.0	4.4	4.4	4.4	6.1	6.1	6.1	6.2	8.8	28.6	29.1	29.7	30.9
132.ijpeg	2.3	2.5	8.4	74.6	76.7	84.6	86.0	99.2	130.2	154.4	165.6	201.5	240.0	205.5	194.7
147.vortex	0.2	0.3	0.3	0.2	0.0	0.1	0.0	0.4	0.8	1.2	2.3	2.5	3.0	4.9	3.7
instf	0.0	0.0	0.0	66.1	100.0	100.9	110.8	115.5	124.0	123.6	121.7	124.7	155.6	170.3	187.0
mulaw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12.4	12.4	49.7
radproc	0.4	0.6	0.6	9.1	13.0	18.7	20.1	30.6	39.2	45.9	50.9	53.2	48.7	54.4	53.5
rfast	0.0	0.0	0.0	159.3	240.1	242.7	266.2	274.0	298.3	291.3	281.3	282.2	342.3	379.5	400.0
rtpse	0.2	0.2	0.2	3.1	8.6	9.0	14.9	32.5	37.2	33.7	32.2	43.5	41.3	60.7	71.6
djpeg	1.4	1.9	8.2	132.1	167.5	195.5	197.4	230.8	244.9	248.7	261.3	277.9	268.0	173.0	172.8
rawcaudio	0.5	0.5	1.0	1.0	2.6	2.6	2.6	2.6	6.0	6.0	5.5	1.1	80.2	44.3	57.5
mpeg2decode	1.2	1.2	1.2	1.5	1.8	1.9	3.3	3.5	4.3	6.2	9.1	9.9	12.8	38.2	94.7
mpeg2encode	0.0	37.2	102.4	97.5	106.6	100.6	93.8	104.1	100.5	105.4	115.0	135.6	137.9	121.0	136.4
unepic	0.3	0.5	2.1	13.3	16.1	23.6	23.9	34.2	38.0	38.9	40.0	60.4	80.8	105.7	105.8
average	1.1	2.5	5.3	20.9	27.2	29.1	30.5	34.8	39.0	40.4	42.4	49.2	59.3	57.5	65.4

Table A.6: Speedup in percentages of integrated assignment compared to dependence-conscious early assignment, using software pipelining and the TTA$_{realistic}$ template.

Benchmark	Number of Registers														
	512	64	48	32	30	28	26	24	22	20	18	16	14	12	10
a68	0.6	0.6	0.6	1.1	1.2	1.1	1.7	1.6	1.8	1.4	0.9	1.3	2.2	2.5	3.3
bison	1.5	1.8	1.3	1.2	1.6	1.5	2.2	2.7	2.7	3.8	4.5	4.3	5.5	5.7	1.9
cpp	1.0	1.2	1.2	0.9	1.0	0.4	0.2	0.3	-0.4	0.6	2.1	2.4	3.0	5.4	11.2
crypt	0.0	4.4	4.0	8.3	17.0	15.9	-13.4	-10.9	2.4	-4.5	-2.3	6.2	22.2	26.9	23.1
diff	0.4	0.4	1.1	10.8	9.8	9.6	9.6	10.9	13.5	15.6	15.2	13.9	15.8	19.0	30.1
expand	0.7	2.3	2.8	0.9	3.4	4.5	2.9	2.7	3.2	1.9	-9.1	-6.7	-8.8	-8.0	-11.9
flex	1.0	1.1	1.3	0.6	0.0	0.2	-0.2	0.0	0.5	3.1	1.7	3.8	5.5	6.2	9.9
gawk	1.9	1.9	1.8	1.9	1.9	1.9	1.9	1.8	1.4	2.9	3.4	3.4	2.1	0.9	2.2
gzip	4.6	4.6	4.8	1.9	4.4	4.2	3.4	4.9	5.5	5.3	9.8	10.2	11.6	18.8	28.1
od	0.5	0.7	1.0	0.6	0.6	0.0	1.5	3.1	3.9	7.5	5.0	8.1	8.5	7.0	16.5
sed	-0.5	0.7	0.0	3.8	5.0	5.5	6.3	5.2	6.3	5.6	6.5	8.1	6.4	7.0	11.0
sort	0.4	1.5	1.5	2.5	2.1	1.3	3.3	4.1	7.1	15.6	21.8	40.8	40.1	56.1	52.5
uniq	-0.8	-1.0	-1.3	-0.4	-0.1	1.3	6.6	7.6	6.6	7.5	9.7	11.9	11.3	13.1	21.6
virtex	1.7	1.2	1.6	1.6	1.7	2.1	1.7	2.5	3.6	3.5	4.4	6.3	7.3	10.7	10.2
wc	0.0	0.0	0.0	11.9	11.9	11.9	11.9	11.9	11.9	11.9	9.3	11.1	12.8	5.4	10.3
099.go	1.3	1.2	1.3	1.2	1.2	1.1	1.5	3.0	3.3	2.1	2.2	3.3	5.0	6.5	4.4
124.m88ksim	2.9	3.0	2.8	3.2	3.1	2.9	2.1	2.2	3.0	4.0	3.7	2.8	3.3	-0.6	-5.7
129.compress	0.0	0.0	0.0	4.3	4.3	4.3	5.9	5.9	6.2	6.9	9.0	27.5	28.2	27.5	31.4
132.ijpeg	-0.2	1.1	1.8	6.4	4.6	11.2	31.3	51.9	59.0	53.2	53.0	65.2	65.7	56.2	73.2
147.vortex	0.3	0.5	0.4	0.2	-0.1	0.0	0.3	0.1	0.6	0.9	1.8	1.8	1.8	4.6	-3.0
instf	-2.9	-2.9	-2.9	-2.9	-2.9	-2.9	-2.9	-2.2	-1.3	-2.2	-1.4	2.1	6.3	34.8	41.8
mulaw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
radproc	-2.3	-2.7	-2.4	-2.6	-2.4	-2.3	-1.8	0.5	5.2	5.6	5.0	7.3	9.9	25.3	31.1
rfast	-5.4	-5.4	-5.4	-5.4	-5.4	-5.4	-5.4	-4.7	-2.3	-3.9	-2.2	3.6	11.9	62.3	73.4
rtpse	-0.6	-0.6	-0.6	-0.5	-0.7	-0.7	-0.9	0.4	3.2	7.5	10.5	27.5	23.9	29.8	29.7
djpeg	0.3	0.4	4.6	10.4	5.4	20.1	27.7	36.0	58.7	81.0	69.9	79.6	82.2	66.1	69.4
rawcaudio	0.7	0.7	0.7	0.8	4.7	4.7	4.8	4.8	6.0	5.9	3.7	-2.6	68.4	37.0	56.7
mpeg2decode	0.1	0.1	0.2	0.1	0.1	0.2	0.1	0.4	0.8	2.2	3.6	5.8	8.6	19.1	66.7
mpeg2encode	-1.1	7.5	42.4	65.3	64.1	62.9	60.3	57.1	61.2	67.3	62.4	72.7	71.4	63.8	71.4
unepic	-0.4	-0.4	0.3	-1.2	-0.8	11.3	10.3	7.2	6.3	6.3	11.8	19.8	27.5	49.0	45.3
average	0.2	0.8	2.2	4.2	4.6	5.6	5.8	7.0	9.3	10.6	10.5	14.7	18.7	21.9	26.9

Partitioned Register File Benchmark Results

B

In this appendix the individual benchmark results of the partitioned register file experiments are listed. The numbers indicate the performance loss in percentages relative to a monolithic register file with four read and four write ports. Two monolithic register files are used in the experiments: one with 512 registers and one with 32 registers. In Section B.1 the results of early assignment are listed. Section B.2 lists the results of integrated assignment.

B.1 Early Assignment

This section shows the impact of register file partitioning on the individual benchmark results, when using dependence-conscious early assignment. The results listed in Table B.1 clearly show that the chosen distribution has a large impact on the performance. More sophisticated partitioning methods, as shown in Table B.2, did not result in a further improvement. It should be noted, however, that the impact of register file partitioning is small.

Table B.1: Performance loss in percentages of the vertical and horizontal distribution when the register file of the TTA$_{realistic}$ template is partitioned.

	Vertical distribution				Horizontal distribution			
	2x256	4x128	2x16	4x8	2x256	4x128	2x16	4x8
a68	0.7	20.9	0.4	10.6	0.3	1.4	0.2	1.6
bison	3.3	28.0	0.9	10.2	1.0	2.7	0.6	2.2
cpp	1.2	11.0	1.2	6.3	0.4	0.4	0.5	0.7
crypt	26.0	63.9	15.1	26.6	0.3	8.2	1.6	12.2
diff	4.1	44.6	2.2	16.4	1.2	7.4	0.2	6.8
expand	8.6	41.4	4.8	13.5	3.9	10.5	3.8	6.4
flex	3.8	24.5	2.6	11.6	0.3	3.6	0.6	4.7
gawk	0.4	4.2	0.0	3.2	0.3	0.1	0.2	0.9
gzip	-3.8	20.4	2.1	9.0	0.2	1.0	1.0	3.5
od	14.6	67.2	10.1	22.0	3.9	7.9	1.2	7.0
sed	4.0	26.0	2.4	13.5	0.9	5.1	0.5	5.6
sort	8.2	51.9	3.8	20.9	0.3	2.4	0.7	5.7
uniq	6.6	45.6	5.6	22.8	0.9	5.8	0.3	6.8
virtex	1.9	14.7	0.8	6.6	0.1	2.0	-0.3	1.7
wc	0.0	13.0	0.1	-1.0	0.0	0.3	0.1	-0.6
099.go	2.1	18.0	1.7	7.7	0.2	2.1	0.1	1.4
124.m88ksim	2.1	18.9	2.0	9.6	0.5	2.6	0.4	1.5
129.compress	1.6	12.7	0.3	5.5	0.3	3.0	0.0	1.6
132.ijpeg	17.8	84.1	5.1	18.8	1.8	8.2	2.0	8.2
147.vortex	1.4	33.2	0.6	16.8	-0.2	0.9	0.1	1.4
instf	1.8	24.7	2.7	5.2	0.0	-0.5	2.0	3.6
mulaw	0.5	2.4	0.5	2.4	0.0	0.0	0.0	0.0
radproc	2.3	28.4	1.0	7.9	0.2	1.1	0.0	2.5
rfast	2.1	34.3	0.0	-0.6	0.6	1.8	0.0	0.2
rtpse	2.4	29.5	0.8	1.2	0.1	1.1	0.4	2.2
djpeg	18.8	92.9	5.8	22.9	1.9	9.6	1.5	8.5
rawcaudio	3.9	42.7	0.6	7.4	0.4	6.5	0.3	4.2
mpeg2decode	1.8	13.7	3.7	2.8	-1.2	1.9	2.7	1.2
mpeg2encode	35.0	119.4	4.7	17.7	3.7	14.1	2.7	10.7
unepic	7.1	35.5	1.9	7.6	0.6	2.9	1.2	3.6
average	6.0	35.6	2.8	10.8	0.8	3.8	0.8	3.9

Table B.2: Performance loss in percentages of the data independence heuristic and the intra-operation heuristic when the register file of the TTA$_{realistic}$ template is partitioned.

	Data independence heuristic				Intra-operation heuristic			
	2x256	4x128	2x16	4x8	2x256	4x128	2x16	4x8
a68	0.3	1.7	0.4	1.2	0.3	1.2	0.3	1.4
bison	1.4	3.1	0.8	3.8	0.3	2.4	0.9	2.9
cpp	0.1	2.7	0.2	3.4	1.6	2.1	1.5	0.5
crypt	12.8	28.8	4.1	13.2	0.6	6.1	1.7	11.5
diff	1.6	10.2	-1.9	6.7	0.3	4.4	0.3	6.7
expand	2.6	13.7	5.4	14.1	4.7	12.3	3.3	6.0
flex	0.3	3.2	0.8	4.5	0.4	2.9	0.6	3.8
gawk	0.2	0.8	0.3	0.9	0.0	-0.1	-0.2	0.5
gzip	-3.0	0.9	1.2	6.9	0.2	1.5	0.1	2.1
od	6.6	6.9	3.1	10.1	3.5	7.8	0.7	5.2
sed	2.8	6.3	2.4	6.2	0.8	4.2	0.6	5.7
sort	2.8	11.2	2.5	9.2	2.3	2.6	0.9	6.0
uniq	2.6	15.5	2.0	13.3	0.4	6.9	-0.1	9.0
virtex	1.1	3.4	1.1	2.8	0.5	2.0	0.2	2.3
wc	0.0	3.6	-8.5	-7.1	0.0	0.3	0.1	-0.6
099.go	0.8	1.9	2.6	3.2	0.2	2.1	0.1	1.4
124.m88ksim	0.4	2.8	-0.5	1.0	0.5	2.1	0.2	2.2
129.compress	0.3	0.8	-2.1	0.1	0.3	3.0	0.3	1.6
132.ijpeg	5.1	13.0	4.1	10.8	2.1	7.9	2.3	8.0
147.vortex	0.3	0.6	0.0	1.0	-0.1	0.7	-0.2	1.1
instf	0.1	1.2	1.1	0.8	-0.6	-1.6	-0.5	4.8
mulaw	-2.8	-3.3	-2.8	-3.3	0.0	0.0	0.0	0.0
radproc	0.0	2.1	1.0	2.6	0.5	0.5	0.1	2.5
rfast	0.4	2.0	-1.4	0.6	0.5	1.6	0.0	2.3
rtpse	-0.1	1.5	-0.3	1.6	0.0	1.6	0.4	2.6
djpeg	2.8	12.5	2.1	12.0	1.9	7.2	2.7	9.3
rawcaudio	0.5	1.2	0.8	9.3	0.4	4.1	-0.1	3.7
mpeg2decode	1.5	1.5	3.1	2.4	0.1	1.3	1.4	3.6
mpeg2encode	6.6	18.4	4.3	11.4	8.0	11.7	3.8	11.1
unepic	1.6	3.0	7.5	6.2	1.1	2.7	0.8	3.5
average	1.7	5.7	1.1	5.0	1.0	3.4	0.7	4.0

B.2 Integrated Assignment

This section shows the impact of register file partitioning on the individual benchmark results, when using integrated assignment. The results for the local heuristics (first-fit, best-fit and round robin) are shown in Table B.3. The results for the global heuristic are shown in Table B.4. The tables show that the round robin strategy performs best.

Table B.3: Performance loss in percentages of the local heuristics: first-fit, best-fit and round robin, when the register file of the TTA$_{realistic}$ template is partitioned.

	first-fit heuristic				best-fit heuristic				round robin heuristic			
	2x256	4x128	2x16	4x8	2x256	4x128	2x16	4x8	2x256	4x128	2x16	4x8
a68	3.4	11.2	3.6	6.6	3.3	11.2	3.5	6.6	0.0	0.5	-0.2	0.6
bison	5.4	26.7	2.3	9.7	5.4	26.7	2.3	9.7	1.5	2.4	1.1	1.1
cpp	1.5	10.6	0.8	5.1	1.4	10.6	0.7	5.1	0.1	1.7	-0.7	-0.7
crypt	1.6	25.2	1.2	3.2	1.4	25.2	0.8	3.2	0.1	1.9	0.0	1.9
diff	4.6	30.0	2.7	21.5	3.8	30.0	2.5	21.5	1.1	3.2	0.3	2.7
expand	10.5	28.9	8.3	17.2	10.5	28.9	8.9	17.2	0.6	6.6	0.3	4.1
flex	4.0	17.6	2.2	8.6	2.7	17.6	1.5	8.6	1.0	2.7	1.5	2.3
gawk	1.2	4.1	1.0	4.3	1.2	4.1	1.0	4.3	-0.8	0.6	-0.8	1.0
gzip	0.5	18.2	-0.8	4.0	0.8	18.2	0.4	4.0	-0.3	-0.1	-1.5	-0.6
od	12.9	55.7	10.5	26.3	8.9	55.7	6.7	26.3	1.7	2.7	-1.0	2.3
sed	6.1	24.0	2.3	10.2	4.6	24.0	1.9	10.2	0.9	3.2	-0.1	3.1
sort	7.9	25.0	5.6	14.2	5.8	25.0	4.2	14.2	1.2	4.9	1.0	6.7
uniq	5.4	24.0	4.5	12.7	5.3	24.0	3.8	12.7	-1.1	4.5	-0.6	2.7
virtex	2.2	14.0	0.9	4.8	2.2	14.0	0.9	4.8	0.4	1.8	0.1	2.7
wc	0.0	27.4	0.0	18.0	0.0	27.4	0.0	18.0	0.0	0.8	0.0	-0.2
099.go	1.4	11.8	4.2	8.9	1.4	11.8	3.4	8.9	0.2	2.3	1.2	5.0
124.m88ksim	1.3	12.9	0.5	9.7	1.3	12.9	0.5	9.7	0.2	4.2	0.6	1.2
129.compress	1.3	5.3	1.6	5.8	0.3	5.3	0.3	5.8	0.0	2.0	0.6	1.9
132.ijpeg	13.3	53.3	2.5	15.3	9.2	53.3	1.9	15.3	1.3	6.3	-0.3	3.8
147.vortex	7.4	19.0	6.9	8.6	7.3	19.0	6.9	8.6	0.2	1.1	-0.1	0.7
instf	1.8	12.8	1.1	11.6	0.3	12.8	0.2	11.6	0.0	0.2	0.1	-1.0
mulaw	0.0	-1.4	0.0	0.0	0.0	-1.4	0.0	0.0	0.0	0.0	-1.4	-1.4
radproc	1.1	9.2	1.0	5.8	0.7	9.2	0.5	5.8	0.1	2.7	-0.6	1.9
rfast	2.2	12.3	0.5	7.9	0.8	12.3	0.6	7.9	-0.1	1.8	0.0	0.4
rtpse	2.4	14.7	1.2	8.4	0.9	14.7	0.4	8.4	0.1	1.8	0.1	1.4
djpeg	13.3	52.1	3.3	13.1	10.1	52.1	2.3	13.1	2.5	6.4	-0.2	6.0
rawcaudio	4.0	39.0	0.6	10.7	4.0	39.0	0.6	10.7	1.0	7.4	1.0	8.1
mpeg2decode	1.7	5.8	2.8	2.2	1.5	5.8	2.7	2.2	1.4	0.4	1.4	0.6
mpeg2encode	24.7	80.3	3.6	15.7	22.8	80.3	3.9	15.7	6.5	14.0	1.3	12.3
unepic	5.6	16.0	0.6	5.2	4.1	16.0	-0.6	5.2	0.3	1.5	0.3	2.2
average	5.0	22.9	2.5	9.8	4.1	22.9	2.1	9.8	0.7	3.0	0.1	2.4

Table B.3: Performance loss in percentages of the global heuristic when the register file of the TTA$_{realistic}$ template is partitioned.

Benchmark	RF Partitions			
	2x256	4x128	2x16	4x8
a68	-0.2	0.5	-0.2	0.4
bison	1.5	2.9	0.7	1.9
cpp	-0.8	0.3	0.3	0.4
crypt	0.2	3.2	0.3	2.7
diff	0.9	2.1	0.5	2.1
expand	0.6	6.4	2.5	4.2
flex	1.0	3.1	0.4	3.5
gawk	-0.1	1.3	-0.1	1.5
gzip	-0.4	3.7	-0.8	1.1
od	-0.2	5.4	-0.8	3.7
sed	1.6	4.1	-0.1	3.6
sort	1.5	3.5	1.1	5.3
uniq	-1.0	2.7	-1.3	4.5
virtex	0.4	1.7	0.6	1.9
wc	0.0	0.8	0.0	0.6
099.go	0.2	2.2	1.5	5.5
124.m88ksim	0.3	3.6	0.7	3.8
129.compress	0.0	1.9	0.0	1.0
132.ijpeg	1.8	5.7	-0.6	4.5
147.vortex	0.3	0.6	0.4	0.6
instf	0.0	-0.2	0.1	2.9
mulaw	0.0	0.0	-1.4	-1.4
radproc	0.8	0.2	0.4	1.0
rfast	0.3	0.8	0.1	1.0
rtpse	0.3	3.9	0.2	2.6
djpeg	1.4	7.1	0.6	2.3
rawcaudio	0.8	5.3	0.5	4.8
mpeg2decode	1.3	2.5	1.4	1.1
mpeg2encode	5.0	12.2	1.9	15.0
unepic	0.3	2.2	0.7	1.8
average	0.6	3.0	0.3	2.8

Glossary

This glossary gives an overview of the symbols and abbreviations used throughout this thesis.

Symbols

$\top$	Definition point of all variables v referenced in basic block b and $v \in live_{In}(b)$
$\bot$	Use point of all variables v referenced in basic block b and $v \in live_{Out}(b)$
$\|S\|$	Cardinality (size) of the set S
γ_R	Register rank
γ_S	Schedule rank
δ^a	Anti dependence
δ_a	Percentage dimensional increase per RF-port
δ^f	Flow dependence
$\delta^i_{delay,distance}$	Inter-iteration data dependence edge with a minimal delay of $delay$ instructions and a minimal distance of $distance$ iterations
δ^o	Output dependence
δ^{ot}	Operand-trigger dependence
δ^{tr}	Trigger-result dependence
δ^{type}_{delay}	Data dependence edge of type $type$ with a minimal de-

	lay of $delay$ instructions
ρ_p	Relative access time increase per RF-port
ρ_r	Relative access time increase per register
$Area_{RF}$	Relative area of a register file
b	A basic block
b^*	Currently scheduled basic block
$bb(n)$	Basic block of operation n
B	Set of basic blocks
$B_{DU}(v)$	Set of basic blocks containing references to variable v
$B_{IO}(v)$	Set of basic blocks in which variable v is live on exit and entry
BE	Set of back edges
$C(G)$	Transitive closure of a graph G
$C_{\textit{callee-saved}}(v)$	Callee-saved cost of variable v
$C_{\textit{caller-saved}}(v)$	Caller-saved cost of variable v
$C_{spill}(v)$	Spill cost of variable v
$C_{\textit{state preserving}}(v, r)$	State preserving cost when register r is mapped onto variable v
CE	Set of control flow edges
$degree(n)$	Number of neighbor nodes of a node n
$delay(o_j, o_i)$	Data dependence delay between the operations o_j and o_i
$b_i\ doms\ b_j$	Dominance relation between basic blocks b_i and b_j
D	Set of duplication basic blocks
E_{du}	Set of du-chain edges
E_{DDG}	Set of DDG edges
E_f	Set of false dependence edges
$E_{\textit{fdp}}$	Set of false dependence prevention edges
$E_{\textit{ff}}$	Set of forward false dependence edges
$E_{\textit{ffdp}}$	Set of forward false dependence prevention edges
$E_{\textit{fpar}}$	Set of forward parallel interference edges
$E_{\textit{interf}}$	Set of interference edges
E_{Mem}	Set of memory data dependence edges
E_{min}	Set of minimal interference edges
E_{par}	Set of parallel interference edges
E_{Reg}	Set of register data dependence edges
E_{RF}	Average access energy per instruction of an RF
$E_{\textit{RF-ports}}$	Set of RF-port interference edges
E_{SSG}	Set of interference edges in IG_{SSG}
E_t	Set of undirected edges of the transitive closure of the DDG
E^T	Set of edges of the transitive closure of the DDG

E_{Var}	Set of variable data dependence edges
$f()$	Execution frequency
fu	A function unit
FE	Set of forward edges
FU_{set}	Set of function units
G_f	False dependence graph
G_{ff}	Forward false dependence graph
G_t	Transitive closure of the DDG with undirected edges
i	An instruction
i_{cur}	Instruction in which a move is currently being scheduled
interf(v)	Set of variables interfering with variable v
I	Set of intermediate basic blocks
IG	Interference graph
IG_{fpar}	Forward parallel interference graph
IG_{min}	Minimal interference graph
IG_{par}	Parallel interference graph
$IG_{RF\text{-}ports}$	RF-port interference graph
IG_{SSG}	Scheduler-sensitive global interference graph
$lr(v)$	Live range of variable v
$live_{Def}(b)$	Set of variables defined in basic block b before they are used in b
$live_{In}(b)$	Set of live variables at the entry of a basic block b
$live_{Out}(b)$	Set of live variables at the exit of a basic block b
$live_{Use}(b)$	Set of variables used in basic block b before they are defined in b
$live(Q)$	Set of variables live at a point Q
$L(o)$	Latency of operation o
$L_{path}(o_i, o_j)$	Distance in the DDG between the operations o_i and o_j
$L(v)$	Length of the live range of variable v
m	A move (transport)
$m_{def(v)}$	Move defining variable v
m_i	Move with index i
m_o	An operand move
m_r	A result move
$m_{ref(v)}$	Move referring to variable v
m_t	A trigger move
n	A node in a graph
$n_{def(v)}$	Node in the DDG that defines variable v
n_i	Node with index i
$n_{ref(v)}$	Node in the DDG that references variable v

$n_{use(v)}$	Node in the DDG that uses variable v
N_{du}	Nodes of a du-chain
$N_{Def(v)}$	Set of moves that define variable v
$N_{Def(v,b)}$	Set of moves that define variable v in basic block b
N_{DDG}	Set of nodes in a DDG
$N_{Use(v)}$	Set of moves that use variable v
$N_{Use(v,b)}$	Set of moves that use variable v in basic block b
N_{ports}	Number of RF-ports
N_{var}	Set of variables
o	An operation
o_i	Operation with index i
$ot_{freedom}$	Upperbound of the operand-trigger scheduling distance
$O_{\perp}$	Set of ready operations that end a live range
$pred(m)$	Set of predecessor moves of m in the DDG
$pred^*(o)$	Set of scheduled predecessor operations of o in the DDG
$\mathcal{P}$	A program
P	A procedure
$Pr\,(b'\|b)$	Probability that b' will be executed after b
r	A register
$ready$	Set of ready operations
`ri`	Register with index i
$r(v)$	Returns register r mapped onto variable v
R	Set of registers in the target TTA
$R_{\textit{callee-saved}}$	Set of callee-saved registers
$R_{\textit{caller-saved}}$	Set of caller-saved registers
$R_{Cur}(v,n)$	Instruction precise interfering register set for variable v in a partly scheduled basic block $bb(n)$
$R_{DU}(v,b)$	Set of interfering registers of variable v in a basic block $b \in B_{DU}(v)$
R_{free}	Set of available registers in the last instruction in the currently scheduled basic block
$R_{\textit{Interfere}}(v,n)$	Set of interfering registers for a variable v
$R_{IO}(v,b)$	Set of interfering registers of variable v in a basic block $b \in B_{IO}(v)$
$R_{\textit{Non-interfere}}(v,n)$	The set of non-interfering registers for a variable v
R_{RF}	Number of registers in a register file
$R_{RRV}(v,n)$	Set of interfering registers based on RRV information
$RF(r_i)$	Register file of register r_i
$succ_v(o_i)$	Set of successor operations of o_i in the DDG that read variable v

S	A schedule
t_{exec}	Execution time
$tr_{freedom}$	Upperbound of the trigger-result scheduling distance
T_{access}	Relative access time of a multi-ported register file
$T_{path}(o_i, o_j)$	Total path length in the DDG of all paths from o_i to o_j
v	A variable
`vi`	Variable with index i
$V_{Above}(b, v)$	Set of non-interfering variables variables whose live range ends before the live range of v starts in basic block b
$V_{Around}(b, v)$	Set of non-interfering variables that are live at entry and exit of basic block b, and are refined and used within b and do not interfere with v
$V_{Below}(b, v)$	Set of non-interfering variables whose live range starts after the end of the live range of v in basic block b
$V_{Between}(b, v)$	Set of non-interfering variables defined after all uses of v, and whose live range ends before the definition of v
$V_{non\text{-}interf}(b, v)$	Set of variables that do not interfere with v in basic block b under any legitimate schedule
$W_{fffdp}(v_i, v_j)$	Weight associated with a forward false dependence prevention edge

Abbreviations

ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASP	Application Specific Processor
BOP	Billions of Operations Per Second, Inc.
BSD	Berkeley Software Distribution
CFA	Control Flow Analysis
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CNS	Controlled Node Splitting
CPU	Central Processing Unit
CRAIG	Combining Register Assignment Interference Graphs
DAG	Directed Acyclic Graph
DCEA	Dependence Conscious Early Assignment
DDA	Data Dependence Analysis
DDG	Data Dependence Graph
DEC	Digital Equipment Corporation

DFA	Data Flow Analysis
DSP	Digital Signal Processing
du-chain	Definition-use chain
EPIC	Explicitly Parallel Instruction Computing
FDPG	False Dependence Prevention Graph
FFT	Fast Fourier Transform
FIFO	First In First Out
FU	Function Unit
GSC	Global Spill Cost heuristic
HLL	High Level Language
HRMS	Hypernode Reduction Modulo Scheduling
IBM	International Business Machines
IG	Interference Graph
II	Initiation Interval
ILP	Instruction-Level Parallelism
IO	Input/Output
IPS	Integrated Prepass Scheduling
IW	Issue Width
JPEG	Joint Photographic Experts Group
MII	Minimum Initiation Interval
MPEG	Moving Pictures Expert Group
NEC	Nippon Electric Company
NP	Nondeterministic-Polynomial
OTA	Operation Triggered Architecture
PC	Personal Computer
PDA	Personal Digital Assistant
RASE	Register Allocation with Schedule Estimates
RF	Register File
RISC	Reduced Instruction Set Computer
RRV	Register Resource Vector
SCP	Single Copy on a Path rule
SFU	Special Function Unit
SPEC	Standard Performance Evaluation Corporation
SRAM	Static Random Access Memory
TLB	Translation-Lookaside Buffer
TTA	Transport Triggered Architecture
TV	TeleVision
URSA	Unified ReSource Allocator
VLIW	Very Long Instruction Word architecture or processor
VTL	Virtual-time latching

Bibliography

Abe00. Aberdeen Group, Inc. *Intel's Itanium: Who Benefits from Early Adoption*, Aberdeen Group, Inc., Boston, November 2000.

ACM+98. D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B-C. Cheng, P.R. Eaton, Q.B. Olaniran, and W-M.W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, Barcelona, Spain, June 1998.

AEBK94. W. Ambrosch, M.A. Ertl, F. Beer, and A. Krall. Dependence-Conscious Global Register Allocation. In *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 125–136, Zürich, Switzerland, March 1994.

AHC96. H. Adriani, F. Harmsze, and H. Corporaal. The Utilization of a Fully Configurable Microprocessor Development Environment for Rapid VHDL Prototyping and Implementation of 'C'-Based Algorithms. In *Proceedings of ProRisc Workshop*, Mierlo, The Netherlands, November 1996.

Arn01. M.L.C.H. Arnold. *Instruction Set Extensions for Embedded Processors*. PhD thesis, Delft University of Technology, The Netherlands, March 2001.

ASU85. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

BCK91. D. Bernstein, D. Cohen, and H. Krawczyk. Code Duplication: An Assist for Global Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 103–113, Albuquerque, New Mexico, November 1991.

BCKT89. P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 275–284, Portland, Oregon, June 1989.

BCT94. P. Briggs, K.D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

BEH91a. D.G. Bradlee, S.J. Eggers, and R.R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System*, pages 122–131, Santa Clara, California, April 1991.

BEH91b. D.G. Bradlee, S.J. Eggers, and R.R. Henry. The Marion System for Retargetable Instruction Scheduling. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 229–240, Toronto, Canada, June 1991.

BGM+89. D. Bernstein, M.C. Golumbic, Y. Mansour, R.Y. Pinter, D.Q. Goldin, H. Krawczyk, and I. Nahshon. Spill Code Minimization Techniques for Optimizing Compilers. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 258–263, Portland, Oregon, June 1989.

BGS93. D. Berson, R. Gupta, and M.L. Soffa. URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures. In *Proceedings of the Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, January 1993.

BGS94. D. Berson, R. Gupta, and M.L. Soffa. Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers. In *Proceedings of the 1994 Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, Montreal, Canada, August 1994.

BR91. D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, Canada, June 1991.

Bri92. P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice

University, April 1992.

BS91. M. Breternitz and J.P. Shen. Implementation Optimization Techniques for Architecture Synthesis of Application-Specific Processors. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 114–123, Albuquerque, New Mexico, November 1991.

BSBC95. T.S. Brasier, P.H. Sweany, S.J. Beaty, and S. Carr. CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment. In *Proceedings of the 1995 Conference on Parallel Architectures and Compilation Techniques*, pages 11–18, Limassol, Cyprus, June 1995.

BYA93. G.R. Beck, D.W.L. Yen, and T.L. Anderson. The Cydra 5 Minisupercomputer: Architecture and Implementation. *The Journal of Supercomputing*, 7(1/2):143–180, May 1993.

CAC+81. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47–57, January 1981.

CBLM96. T.M. Conte, S. Banerjia, S.Y. Larin, and K.N. Menezes. Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 201–211, Paris, France, December 1996.

CC97. A. G. M. Cilio and H. Corporaal. Efficient Code Generation for ASIPs with Different Word Sizes. In *Proceedings of the third Annual Conference of the Advanced School for Computing and Imaging*, pages 144–150, Heijen, The Netherlands, June 1997.

CCK97. C.M. Chang, C.M. Chen, and C.T. King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-issue Processors. *Computers and Mathematics with Applications*, 34(9):1–14, 1997.

CDN93. A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1993.

CDN94. A. Capitanio, N. Dutt, and A. Nicolau. Toward Register Allocation for Multiple Register File VLIW Architectures. Technical Report TR 6-94, University of California, Irvine, Department of Information and Computer Science, February 1994.

CDN95. A. Capitanio, N. Dutt, and A. Nicolau. Architectural Tradeoff Analysis of Partitioned VLIWs. Technical Report TR 14-94, University of California, Irvine, Department of Information and Computer Sci-

ence, April 1995.

CF87. R. Cytron and J. Ferrante. What's in a Name? or The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

CGVT00. J-L. Cruz, A. González, M. Valero, and N.P. Topham. Multiple-banked Register File Architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, Vancouver, Canada, June 2000.

CH90. F.C. Chow and J.L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–526, October 1990.

CH95. H. Corporaal and J. Hoogerbrugge. Code Generation for Transport Triggered Architectures. In Gert Goossens and Peter Marwedel, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

Cha82. G.J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 1982.

CJA00. H. Corporaal, J.A.A.J. Janssen, and M.L.C.H. Arnold. Computation in the context of Transport Triggered Architectures. *International Journal of Parallel Programming*, 28(4):401–427, August 2000.

CK91. D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, Canada, June 1991.

CL95. H. Corporaal and R. Lamberts. TTA Processor Synthesis. In *Proceedings of the first Annual Conference of the Advanced School for Computing and Imaging*, pages 18–27, Heijen, The Netherlands, May 1995.

CLM+95. P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors. *IEEE Transactions on Computers*, 44(3):353–370, March 1995.

CM69. J. Cocke and R. E. Miller. Some Analysis Techniques for Optimizing Computer Programs. In *Proceedings of the 2nd Hawaii Conference on System Sciences*, pages 143–146, Honolulu, Hawaii, January 1969.

CM91. H. Corporaal and J.M. Mulder. MOVE: A Framework for High-Performance Processor Design. In *Supercomputing'91 Proceedings*, pages 692–701, Albuquerque, New Mexico, November 1991.

CND95. A. Capitanio, A. Nicolau, and N. Dutt. A Hypergraph-Based Model for Port Allocation on Multiple-Register-File VLIW Architectures. *International Journal of Parallel Programming*, 23(6):499–513, December 1995.

Cor93. H. Corporaal. Evaluating Transport Triggered Architectures for Scalar Applications. *Microprocessing and Microprogramming*, 38:45–52, September 1993.

Cor98. H. Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley & Sons, 1998.

CS95. T.M. Conte and S.W. Sathaye. Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 208–218, Ann Arbor, Michigan, November 1995.

CvdA93. H. Corporaal and P. van der Arend. MOVE32INT, a Sea of Gates Realization of a High Performance Transport Triggered Architecture. *Microprocessing and Microprogramming*, 38:53–60, September 1993.

DA93. A. Dolan and J. Aldous. *Networks and Algorithms: An Introductory Approach*. John Wiley & Sons, 1993.

dM94. G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

DT93. J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.

EA97. K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, Colorado, June 1997.

ED95a. A.E. Eichenberger and E.S. Davidson. Register Allocation for Predicated Code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, Ann Arbor, Michigan, November 1995.

ED95b. A.E. Eichenberger and E.S. Davidson. Stage Scheduling: A Technique to Reduce Register Requirements of a Modulo Schedule. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, Ann Arbor, Michigan, November 1995.

EFK+98. K. Ebcioglu, J. Fritts, S. Kosonocky, M. Gschwind, E. R. Altman, K. Kailas, and T. Bright. An Eight-Issue Tree-VLIW Processor for Dynamic Binary Translation. In *Proceedings of the International Conference on Computer Design*, pages 488–495, Austin, Texas, October 1998.

EGS95. C. Eisenbeis, F. Gasperoni, and U. Schwiegelshohn. Allocating Registers in Multiple Instruction-Issuing Processors. In *Proceedings of the 1995 Conference on Parallel Architectures and Compilation Techniques*, pages 290–293, June 1995.

Ell86. J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. ACM Doctoral Dissertation Awards. MIT Press, Cambridge, Massachusetts, 1986.

Emb95. P.M. Embree. *C Language Algorithms for Real-Time DSP*. Prentice-Hall, 1995.

EZ97. Embedded Software Onderzoek in Nederland, Analyse en resultaten. Ministerie van Economische Zaken, August 1997.

FBF+00. P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood. LX: A Technology Platform for Customizable VLIW Embedded Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 203–213, Vancouver, Canada, June 2000.

FCJV97. K.I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, Research Triangle Park, North Carolina, December 1997.

FFD96. J.A. Fisher, P. Faraboschi, and G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 324–335, Paris, France, December 1996.

Fis81. J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

FJC95. K.I. Farkas, N.P. Jouppi, and P. Chow. Register File Design Considerations in Dynamically Scheduled Processors. Technical Report WRL 95/10, Digital Western Research Laboratory, Palo Alto, California, November 1995.

FR91. S.M. Freudenberger and J.C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation - Concepts, Tools, Techniques*, pages 146–172, 1991.

GJ79. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

GJS96. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts,

1996.

GL95. K.J. Gough and J. Ledermann. Register Allocation in the Gardens Point Compilers. In *Proceedings 18th Australian Computer Science Conference*, Adelaide, Australia, January 1995.

GNAB92. Jeffrey Gray, Andrew Naylor, Arthur Abnous, and Nader Bagherzadeh. VIPER: A 25-MHz, 100 MIPS Peak VLIW Microprocessor. Technical Report ICS-TR-92-78, University of California, Irvine, Department of Information and Computer Science, July 1992.

GS90. R. Gupta and M.L. Soffa. Region Scheduling: An approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

GS95. J.R. Gurd and D.F. Snelling. Self-Regulation of Workload in the Manchester Data-Flow Computer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 135–145, Ann Arbor, Michigan, November 1995.

GSS89. R. Gupta, M.L. Soffa, and T. Steele. Registers Allocation via Clique Separators. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 264–274, Portland, Oregon, July 1989.

GV97. C. J. Glossner and S. Vassiliadis. The Delft-Java Engine. *Lecture Notes in Computer Science*, 1300:766–770, August 1997.

GWC88. J.R. Goodman and W. Wei-Chung. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the International Conference on Supercomputing*, pages 442–452, St. Malo, France, July 1988.

Gwe96. L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14):11–16, October 1996.

HA99. J. Hoogerbrugge and L. Augesteijn. Instruction Scheduling for TriMedia. *Journal of Instruction-Level Parallelism (http://www.jilp.org/vol1)*, February 1999.

HC89. W.W. Hwu and P. P. Chang. Inline Function Expansion for Compiling C Programs. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 246–257, Portland, Oregon, June 1989.

HC94. J. Hoogerbrugge and H. Corporaal. Register File Port Requirements of Transport Triggered Architectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 191–195, San Jose, California, November 1994.

HC95. J. Hoogerbrugge and H. Corporaal. Automatic Synthesis of Transport Triggered Architectures. In *Proceedings of the first Annual Conference of the Advanced School for Computing and Imaging*, pages 79–88, Heijen, The Netherlands, May 1995.

HC96. J. Hoogerbrugge and H. Corporaal. Resource Assignment in a Compiler for Transport Triggered Architectures. In *Proceedings of the second Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 1996.

Hec77. M.S. Hecht. *Flow Analysis of Computer Programs*. Programming Languages Series. Elsevier, North-Holland, 1977.

HG83. J.L. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

HHR95. R.E. Hank, W-M.W. Hwu, and B.R. Rau. Region-Based Compilation: An Introduction and Motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, Ann Arbor, Michigan, November 1995.

HMC+93. W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1/2):229–248, May 1993.

Hoo96. J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, The Netherlands, February 1996.

HP90. J.L. Hennessy and D.A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann publishers, San Mateo, California, 1990.

Huf93. R.A. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 258–267, Albuquerque, New Mexico, June 1993.

IA99. IA-64 Architecture Innovations. 1999 Intel Developer Forum, Joint Whitepaper between Intel & HP, February 1999.

Int00. Intel Corporation. *Inside the NetBurst^{TM} Micro-Architecture of the Intel Pentium 4 Processor*, Intel Corporation, Santa Clara 2000.

Jan01. I. Janssen. Enhancing the Move Framework. Master's thesis, Delft University of Technology, The Netherlands, May 2001.

JC95. J.A.A.J. Janssen and H. Corporaal. Partitioned Register Files for

TTAs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 303–312, Ann Arbor, Michigan, November 1995.

JC96. J.A.A.J. Janssen and H. Corporaal. Controlled Node Splitting. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 44–58, Linkoping, Sweden, April 1996.

JC97a. J.A.A.J. Janssen and H. Corporaal. Making Graphs Reducible with Controlled Node Splitting. *ACM Transactions on Programming Languages and Systems*, 19(6):1031–1052, November 1997.

JC97b. J.A.A.J. Janssen and H. Corporaal. Registers On Demand: Integrated Register Allocation and Instruction Scheduling . In *Proceedings of the third Annual Conference of the Advanced School for Computing and Imaging*, pages 61–67, Heijen, The Netherlands, June 1997.

JC98. J.A.A.J. Janssen and H. Corporaal. Registers On Demand, an Integrated Region Scheduler and Register Allocator. In *Poster Proceedings of the 7th International Conference on Compiler Construction*, pages 44–51, Lisbon, Portugal, April 1998.

JCK98. J.A.A.J. Janssen, H. Corporaal, and I. Karkowski. Integrated Register Allocation and Software Pipelining. In *Proceedings of the fourth Annual Conference of the Advanced School for Computing and Imaging*, pages 80–86, Lommel, Belgium, June 1998.

Joh91. W.M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

Jol91. R.D. Jolly. A 9-ns, 1.4-Gigabyte/s, 17-Ported CMOS Register File. *IEEE Journal of Solid-State Circuits*, 26(10):1407–1412, October 1991.

JW89. N.P. Jouppi and David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the 3th International Conference on Architectural Support for Programming Languages and Operating System*, pages 272–282, Boston, Massachusetts, April 1989.

Kar72. R.M. Karp. Reducibility among Combinatorial Problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103, Yorktown Heights, New York, March 1972.

Kla00. A. Klaiber. *The Technology Behind Crusoe^{TM} Processors*. Transmeta Corporation, Santa Clara, January 2000.

L^{+}93. P.G. Lowney et al. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1/2):51–142, May 1993.

Lam88. M. S. Lam. Software Pipelining: an Effective Scheduling Technique for VLIW Machines. In *Proceedings of ACM SIGPLAN'88 Conference*

on Programming Language Design and Implementation, pages 318–328, Atlanta, Georgia, June 1988.

Lam98. M. S. Lam. *A Systolic Array Optimizing Compiler*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1998.

Lee92. C.G. Lee. *Code Optimizers and Register Organizations for Vector Architectures*. PhD thesis, University of California, Berkeley, May 1992.

Leh00. R. Lehrbaum. Embedded systems news: Soc it to 'ya! *Linux Journal*, pages 36–37, August 2000.

LG95. L.A. Lozano and G.R. Gao. Exploiting Short-Lived Variables in Superscalar Processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, Ann Arbor, Michigan, December 1995.

LGAV96. J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, Boston, Massachusetts, October 1996.

LPMS97. C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, North Carolina, December 1997.

LVA96. J. Llosa, M. Valero, and E. Ayguadé. Heuristics for Register-Constrained Software Pipelining. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 250–261, Paris, France, December 1996.

LVAG95. J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode Reduction Modulo Scheduling. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 350–360, Ann Arbor, Michigan, November 1995.

LW92. M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992.

LW97. H. Liao and A. Wolfe. Available Parallelism in Video Applications. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 321–329, Research Triangle Park, North Carolina, December 1997.

Mah96. S.A. Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana-

Champaign, 1996.

ME92. S. Moon and K. Ebcioglu. An Efficient Resource Constrained Global Scheduling Technique for Superscalar and VLIW processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, Portland, Oregon, December 1992.

Mes01. B. Mesman. *DSP Code Generation*. PhD thesis, Eindhoven University of Technology, The Netherlands, May 2001.

MLC+92. S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, Oregon, December 1992.

MLD92. P. Michel, U. Lauther, and P. Duzy, editors. *The Synthesis Approach to Digital System Design*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1992.

MPSR95. R. Motwani, K. Palem, V. Sarkar, and S. Reyen. Combined Instruction Scheduling and Register Allocation. Technical Report 698, New York University, Department of Computer Science, July 1995.

Muc97. S.S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

Mue92. F. Mueller. Register Allocation by Graph Coloring: A Review. Technical Report FALL 1992, Florida State University, Department of Computer Science, 1992.

NG93. Q. Ning and G.R. Gao. A Novel Framework of Registers Allocation for Software Pipelining. In *conference record 20th ACM Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, 1993.

NP93. C. Norris and L.L. Pollock. A Scheduler-Sensitive Global Register Allocator. In *Supercomputing'93 Proceedings*, pages 804–813, Portland, Oregon, November 1993.

NP95. C. Norris and L.L. Pollock. An Experimental Study of Several Cooperative Register Allocation and Instruction Scheduling Strategies. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 169–179, Ann Arbor, Michigan, November 1995.

Pin93. S.S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 248–257, Albuquerque, New Mexico, June 1993.

PJS96. S. Palacharla, N.P. Jouppi, and J.E. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328, University of Wisconsin, Madison, Computer Sciences Department, November 1996.

PSM97. S. Park, S. Shim, and S-M Moon. Evaluation of Scheduling Techniques on a SPARC-Based VLIW Testbed. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 104–113, Research Triangle Park, North Carolina, December 1997.

PV00. G.G. Pechanek and S. Vassiliadis. The ManArray Embedded Processor Architecture. In *Proceedings of the 26th Euromicro Conference: "Informatics: inventing the future*, pages 384–355, Maastricht, The Netherlands, September 2000.

Rau93. B.R. Rau. Dynamically Scheduled VLIW Processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 80–92, Austin, Texas, December 1993.

Rau94. B.R. Rau. Iterative Modulo Scheduling: An algorithm For Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, December 1994.

Rau96. B.R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Programming*, 24(1):3–64, February 1996.

RCL01. S. Roos, H. Corporaal, and R. Lamberts. Clustering on the Move. *Submitted for publication*, 2001.

RF93. B.R. Rau and J.A. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective. *The Journal of Supercomputing*, 7(1/2):9–50, May 1993.

RJSS97. E.R. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, North Carolina, December 1997.

RYYT89. B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra 5 Departmental Supercomputer. Design Philosophies, Decisions and Tradeoffs. *IEEE Computer*, 22(1):12–35, January 1989.

SB90. P. Sweany and S. Beaty. Post-Compaction Register Assignment in a Retargetable Compiler. In *Proceedings of the 23th Annual International Symposium on Microarchitecture*, pages 107–116, Orlando, Florida, December 1990.

SBV95. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Com-*

puter Architecture, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

SCD+97. M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang, and C. Thompson. Compilers for Instruction-Level Parallelism. *IEEE Computer*, 30(12):63–69, December 1997.

SM95. A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–392, Los Alamitos, California, November 1995.

Spe96. Spec95. The Standard Performance Evaluation Corporation. (http://www.specbench.org), December 1996.

SRD96. G. A. Slavenburg, S Rathnam, and H. Dijkstra. The TriMedia TM-1 PCI VLIW Mediaprocessor. In *Hot Chips 8*, Stanford, California, August 1996.

SS93. M. Schuette and J. Shen. Instruction-Level Experimental Evaluation of the Multiflow TRACE 14/300 VLIW Computer. *The Journal of Supercomputing*, 7(1/2):249–271, May 1993.

Sta94. R.M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, 1994.

STK98. E. Stümpel, M. Thies, and U. Kastens. VLIW Compilation Techniques for Superscalar Architectures. In *Poster Proceedings of the 7th International Conference on Compiler Construction*, pages 234–248, Lisbon, Portugal, April 1998.

SW94. J.E. Smith and S. Weiss. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. *IEEE Computer*, 14(3):46–58, June 1994.

TGH92. K.B. Theobald, G.R. Gao, and L.J. Hendren. On the Limits of Program Parallelism and its Smootability. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 10–19, Portland, Oregon, December 1992.

TI99. Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, Texas Instruments, Dallas, March 1999.

TNO99. TNO Physics and Electronics Laboratory. *µLog-x Low Power Datalogger ASIC family for complete datalogging system solutions*. TNO Physics and Electronics Laboratory, The Hague, The Netherlands, 1999.

TS88. M.R. Thistle and B.J. Smith. A Processor Architecture for Horizon. In *Supercomputing'88 Proceedings*, pages 35–41, Orlando, Florida, November 1988.

vdG98. A. J. van de Goor. *Testing Semiconductor Memories: Theory and Prac-*

tice. ComTex Publishing, Gouda, The Netherlands, 1998.

VLW00. S. Virtanen, J. Lilius, and T. Westerlund. A Processor Architecture for the TACO Protocol Processor Development Framework. In *Proceedings of the 18th IEEE NorChip Conference*, Turku, Finland, November 2000.

Wal91. D.W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System*, pages 176–188, Santa Clara, California, April 1991.

WFW+95. R. Wilson, R. Franch, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. *An Overview of the SUIF Compiler System*. (http://suif.stanford.edu/suif/suif.html), 1995.

WHB92. N.J. Warter, G.E. Haab, and J.W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, Portland, Oregon, December 1992.

Wil51. M.V. Wilkes. The Best Way to Design an Automatic Calculating Machine. In *Proceedings of the Manchester University Computer Inaugural Conference*, pages 16–18, Manchester, England, July 1951.

WM95. R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

Wol99. A. Wolfe. Analysis: VLIW chips will depend on smarter software. *EE Times Online*, May 1999.

WRP92. T. Wada, S. Rajan, and S.A. Przybylski. An Analytical Access Time Model for On-chip Cache Memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.

ZK97. V. Zyuban and P. Kogge. The Energy Complexity of Register Files. Technical Report TR 97-20, University of Notre Dame, Notre Dame, Indiana, CSE Department, December 1997.

ZK98. V. Zyuban and P. Kogge. The Energy Complexity of Register Files. In *International Symposium on Low Power Electronics and Design*, pages 305–310, Monterey, California, August 1998.

ZK00. V. Zyuban and P. Kogge. Optimization of High-Performance Superscalar Architectures for Energy Efficiency. In *International Symposium on Low Power Electronics and Design*, pages 84–89, Rapallo, Italy, July 2000.

Samenvatting

Compiler Strategieën voor Transport Triggered Architecturen

Johan Janssen

In dit proefschrift worden compiler strategieën gepresenteerd die de aanwezige hardware in processoren efficiënt benutten. Voor het onderzoek is gebruik gemaakt van processoren gebaseerd op de Transport Triggered Architectuur (TTA). Deze door de Technische Universiteit Delft ontwikkelde architectuur is uitermate geschikt voor toepassing in applicatie specifieke processoren. De hardware van de TTA is eenvoudig en modulair van opzet. Daardoor is het eenvoudig om de componenten te dupliceren teneinde de aanwezige rekenkracht te vergroten. De compiler is verantwoordelijk voor het toewijzen van hardware aan operaties en de executie volgorde van de operaties. De prestatie kan worden verhoogd door operaties parallel uit te voeren. De mate van uitbuitbaar instruction-level parallellisme (ILP) wordt beperkt door de hoeveelheid hardware en door de noodzakelijke volgorde van operaties.

Het genereren van een executie volgorde voor operaties wordt instruction scheduling genoemd. In deze fase worden operaties toegewezen aan instructies en worden alle hardware componenten, behalve registers, toegewezen aan de operaties. De registers worden vaak in een aparte fase toegekend. De registers bevatten de waarden van de variabelen in een applicatie. Het toewijzen van een register aan een variabele betekent impliciet het toewijzen van een register aan alle operaties die deze variabele gebruiken. Een extra complicatie treedt op als er onvoldoende registers zijn om alle variabelen te bevatten. In dit geval worden de waarden van de variabelen, aan wie geen register is toegekend, weggeschreven naar het geheugen en indien nodig weer teruggelezen. Registers worden als de moeilijkst toewijsbare hardware componenten beschouwd.

Als instruction scheduling en register toewijzing uitgevoerd worden als aparte fasen, zullen beslissingen die genomen worden door de ene fase invloed hebben op de andere fase. In dit proefschrift zijn verschillende strategieën geëvalueerd. De eerste strategie, vroege register toewijzing, voert eerst register toewijzing uit gevolgd door instruction scheduling. Omdat hetzelfde register toegewezen kan worden aan verschillende variabelen ontstaan er zogenaamde valse afhankelijkheden. Deze valse afhankelijkheden beperken de instruction scheduler in het zo efficiënt mogelijk afbeelden van de ILP van het programma op de ILP van de processor. Er bestaan verschillende strategieën die dit proberen te voorkomen door potentiële valse afhankelijkheden te betrekken bij de register toewijzing. Echter, indien er niet genoeg registers beschikbaar zijn, is het onvermijdelijk dat sommige valse afhankelijkheden toch worden geïntroduceerd. Voor TTAs heeft vroege register toewijzing nog een ander nadeel. Omdat TTAs data direct van de ene functie unit (FU) naar een ander FU kunnen transporteren, is het niet altijd meer noodzakelijk om deze data op te slaan in een register. Dit is echter pas bekend na instruction scheduling. Vroege register toewijzing kan geen gebruik maken van deze optimalisatie en introduceert meer valse afhankelijkheden en geheugentransporten dan strikt noodzakelijk.

De tweede strategie, late register toewijzing, voert eerst instruction scheduling uit gevolgd door register toewijzing. Dit heeft als voordeel dat tijdens instruction scheduling geen nadelige invloed wordt ervaren van valse afhankelijkheden. Bovendien is nu tijdens de register toewijzing bekend welke transporten werkelijk een register nodig hebben. Het nadeel van late register toewijzing is dat operaties vroeger worden geplaatst dan strikt noodzakelijk. Hierdoor neemt het aantal variabelen dat op hetzelfde tijdstip een register nodig heeft toe. Als er onvoldoende registers zijn, moeten sommige waarden naar het geheugen worden geschreven. Dit is problematisch omdat het vrijwel onmogelijk is om extra operaties toe te voegen in parallelle TTA code. In dit proefschrift wordt een oplossing voor dit probleem voorgesteld.

Omdat zowel vroege als late register toewijzing problemen hebben, wordt in dit proefschrift een methode voorgesteld om beide fasen te integreren. De geïntegreerde register toewijzingsmethode is eerst toegepast in combinatie met een eenvoudige instruction scheduler. Naast het correct toewijzen van registers aan variabelen, zijn ook technieken gepresenteerd die gedurende instruction scheduling lees en schrijf operaties naar het geheugen invoegen om bijvoorbeeld het gebrek aan registers te compenseren. Om de methode te valideren is de methode uitgebreid met een agressievere instruction scheduling methode die operaties verplaatst tussen basic blocks met als doel de prestatie te vergroten. Ook is de methode met succes toegepast in combinatie met software pipelining. Software pipelining verplaatst operaties over loop iteratie grenzen met als doel om de ILP tussen loop iteraties te vergroten. Verschillende heuristieken zijn voorgesteld en geëvalueerd om de prestaties verder op te voeren. Voor alle drie instruction scheduling strategieën presteert geïntegreerde

toewijzing beter dan alle andere geëvalueerde register toewijzing strategieën. Met name wanneer het aantal beschikbare registers beperkt is nemen de prestaties aanzienlijk toe.

De registers zijn gegroepeerd in een register file (RF). Het uitbuiten van ILP vereist simultaan toegang tot deze RF door middel van RF-poorten. Het is echter kostbaar om een grote RF te maken met een groot aantal RF-poorten. In dit proefschrift wordt een alternatieve RF architectuur voorgesteld: de gedeelde RF. In deze architectuur wordt de RF opgedeeld in een aantal kleinere RFs. Het blijkt dat, hoewel in totaal hetzelfde aantal registers en RF-poorten beschikbaar zijn, het totale chip oppervlak, de toegangstijd en het energieverbruik afnemen. Het is echter niet meer mogelijk om via een willekeurige RF-poort een willekeurig register te bereiken. Om te voorkomen dat dit tot een grote prestatie vermindering leidt zijn een aantal compiler strategieën gepresenteerd en geëvalueerd. Met eenvoudige heuristieken is het mogelijk om de prestatie vermindering te beperken tot minder dan 5%. Wanneer het bespaarde oppervlak wordt gebruikt voor extra registers en de kleinere toegangstijd in rekening wordt gebracht nemen de prestaties zelfs toe.

Curriculum Vitae

Johan Janssen was born in Wamel on the 5^{th} of March, 1968. After finishing the HAVO at the Pax Christi College in Druten, he started studying in Arnhem in 1985. In 1989 he received his B.S. degree in Technical Computer Science from the Hogeschool Arnhem. Subsequently, he studied Electrical Engineering at the Delft University of Technology, where he joined the Information Theory group of Prof.dr.ir. J. Biemond in 1992. From 1991 until 1992 he was a senior graduate student instructor in analogue electronics. In 1993 he graduated *cum laude* on a master thesis entitled "Calculating Rate-Distortion Functions of Size-Constrained Reproduction Alphabets". Thereafter, from 1994 to 1998, he has been a PhD student at Prof.dr.ir. A.J. van de Goor's Computer Engineering group at the department of Electronic Engineering of the Delft University of Technology, where he performed the research described in this dissertation. In early 1998 he joined TNO's Physics and Electronics Laboratory in the Hague, where he is involved in embedded system design and research towards networked intelligent devices.