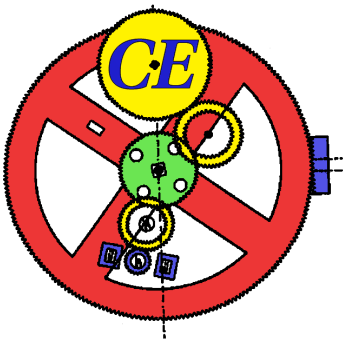


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2013

MSc THESIS

Iterative Instruction Scheduling for a VLIW Processor

Mirshahab Vahedi

CE-MS-2013-03

Abstract

Instruction scheduling aims to reorder instructions in such a way that it covers the delay between an instruction and its dependent successor(s). As a result, the length of schedules are shortened while the processor utilisation increases. This is accomplished by exploiting Instruction Level Parallelism (ILP). The rearrangements made by instruction scheduling plays an important role in achieving the peak performance of a processor, especially for the ones which do not support out-of-order execution.
Optimal scheduling to minimise the number of cycles under an arbitrary pipeline constraints is an NP-complete problem. Hence, most schedulers rely on heuristics in order to arrange the instructions. Although these heuristics are widely used and frequently lead to a fairly good solution, there still might be another instruction order which is better. In this work we bring randomisation to the GNU Compiler Collection (GCC) list scheduler to explore the area of possible orders beyond the heuristics. Our core approach involves *swapping the priorities of instructions*, which does not totally discard the scheduling heuristics. It starts exploring the search space from a fairly good solution obtained by these heuristics. Moreover, as a result of using randomisation in the scheduler, some other problems have been tackled, such as: which part of the search space to explore in a limited amount of time, getting an approximation of how much of the search space is explored, how to fill the delay slots more efficiently, etc.
We evaluated our algorithms in compilation of programs for a Very Long Instruction Word (VLIW) processor called Embedded Vector Processor (EVP) from ST-Ericsson. Since EVP is used as an embedded Digital Signal Processor (DSP) in mobile devices, it is crucial to have a simple architecture to save power. Which is why, EVP is a non-interlocked exposed pipeline and is highly dependent on the compiler to exploit ILP.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Iterative Instruction Scheduling for a VLIW Processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Mirshahab Vahedi
born in Rasht, Iran

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Iterative Instruction Scheduling for a VLIW Processor

by Mirshahab Vahedi

Abstract

Instruction scheduling aims to reorder instructions in such a way that it covers the delay between an instruction and its dependent successor(s). As a result, the length of schedules are shortened while the processor utilisation increases. This is accomplished by exploiting ILP. The rearrangements made by instruction scheduling plays an important role in achieving the peak performance of a processor, especially for the ones which do not support out-of-order execution.

Optimal scheduling to minimise the number of cycles under an arbitrary pipeline constraints is an NP-complete problem. Hence, most schedulers rely on heuristics in order to arrange the instructions. Although these heuristics are widely used and frequently lead to a fairly good solution, there still might be another instruction order which is better. In this work we bring randomisation to the GCC list scheduler to explore the area of possible orders beyond the heuristics. Two approaches have been experimented: *making random decisions* explicitly inside the scheduler's routine, and *swapping the importance of instructions*. The former approach is not making any use of scheduling heuristics, whereas the latter one starts exploring the search space from a fairly good solution obtained by heuristics. Moreover, as a result of using randomisation in the scheduler, some problems have been tackled: which part of the ample search space to explore in a limited amount of time, getting an approximation of how much of the search space is explored, how to fill the delay slots more efficiently, etc.

We evaluated our algorithms in compilation of programs for a VLIW processor called EVP from ST-Ericsson. Since EVP is used as an embedded DSP in mobile devices, it is crucial to have a simple architecture to save power. Which is why, EVP is a non-interlocked exposed pipeline and is highly dependent on the compiler to exploit ILP.

Laboratory : Computer Engineering
Codenumber : CE-MS-2013-03

Committee Members :

Advisor:	Dr. Alex Turjan, ST-Ericsson, Eindhoven
Advisor:	Dr. Anca M. Molnos, CE, TU Delft
Chairperson:	Associate Prof. Sorin D. Cotofana, CE, TU Delft
Member:	Prof. Kees van Berkel, TU Eindhoven
Member:	Associate Prof. Stephan Wong, CE, TU Delft

To my parents and my girlfriend, for supporting me unconditionally.

Contents

List of Figures

List of Algorithms

List of Tables

Acknowledgements

There are so many people who have helped me in this M.Sc thesis that I have decided to have the acknowledgements go three ways.

First, I appreciate the help of my supervisors: I would like to thank Associate Prof. Sorin Cotofana, the advisor of this research, for providing me the opportunity to do research in my field of interest. I would like to appreciate the help of Alex Turjan, my daily supervisor, who constantly guided me through the twists and turns of this work. Thank you Alex, for your valuable feedbacks, and for always opening the path where I was stuck. And, I send very special thanks to my never-tiring academic supervisor, Anca Molnos, who guided me through this thesis and proof read it for so many times.

Second, I am grateful for the staffs at ST-Ericsson for giving me the opportunity to work with them and providing a convenient workplace. I appreciate the help of Claudiu, Dmitry, and Wim, the members of the compiler team, who taught me a lot about the work and how to avoid pitfalls. I thank Arjun, my friend and colleague, for hours of brain storming and discussions.

Finally, I thank my parents for providing me with constant supports of any kind possible, and granting me the chance to be educated at this level. I thank Neda, my fiance and friend, for her never-ending cares and cherishing me when I needed it the most. I would also like to thank Hooman, my true friend, who always kept me going during the hardest times.

Shahab Vahedi
Delft, The Netherlands
February 19, 2013

1 Introduction

1.1 Contexts and Trends

Ever since the advent of Reduced Instruction Set Computers (RISCs) [16] and their pipelined architectures, instruction scheduling techniques have gained importance. Rearranging instructions can *cover* the delay or latency that is required between an instruction and its dependent successor(s).

A significant amount of research conducted in industry and academia has resulted in processors that issue multiple instructions per cycle and hence exploit ILP. Exploiting ILP is considered as a viable approach which provides continuous increase in performance without having to rewrite applications. ILP processors are classified into two broad categories: Very Long Instruction Word (VLIW) and superscalar processors. For VLIW processors the parallelism is exposed at compile time, whereas for superscalar architectures it is exposed at runtime by dynamic instruction scheduling hardware. In VLIW machines, the compiler identifies independent instructions and communicates them to the hardware by packing them in a single long word instruction. However, in a superscalar machine, a complex hardware identifies independent instructions and issues them in parallel at runtime. Hence, compile-time instruction scheduling is solely responsible for exposing and exploiting the parallelism available in a program in a VLIW architecture [20].

The work in this project took place in DSP-IC group of ST-Ericsson which is responsible for designing an embedded VLIW processor, namely the Embedded Vector Processor (EVP). Power consumption plays an important role for embedded processors. Therefore, to have simple design, these processors do not intrinsically support selecting instructions that can be issued simultaneously. Instead, they heavily depend on the compiler to pack parallel instructions together (a very long instruction word) in order to increase throughput. Hence, EVP is a VLIW processor.

In order to compile programs for the EVP, a GCC port is used. It has been observed that some of the programs became faster after using various compilation flags. This speed-up was actually the result of instruction reordering which was a positive side effect of using these flags. As a result of this, it was decided to bring a feature to the compiler that enables it to produce different instruction orderings, *schedules*, while giving it a chance to find better solutions. Since the performance of running programs on EVP is very critical, the DSP-IC group were willing to accept the trade-off between compilation time and the (possible) gain in performance.

The basic block[1] instruction scheduling problem is to find a schedule for a basic block, subject to precedence, latency, and resource constraints [13]. Finding an optimal solution for this problem is NP-complete [1]. One of the dominant algorithms in instruction

[1] A straight-line sequence of code with a single entry point and a single exit point.

scheduling is the *list scheduling*. It is a heuristic-driven approach to construct a schedule one time-slot at a time. While this method finds schedules that are empirically proven to be fairly good, there can be other solutions (that were not produced) which result in shorter clock cycles. One can look at the scheduling problem as a way to search a huge *solution space* (a set of all valid schedules) to find a descent answer.

1.2 Related Work

In order to reduce the length (cycle count) of instruction schedules, many researches have been done. Some of them falls under the category of (randomised) iterative scheduling. In [10], an approach was devised which relied on gradually building a decision tree according to possible choices at each point of time during scheduling. Every path in this tree, from the first choice (root) until the last one (leaf), represents an order of instructions considered for scheduling and thus corresponds to a schedule. This tree can be parsed with different methods like Depth First Search (DFS), Breadth First Search (BFS), and random uniform search. A very simple example of such approach is depicted in Figure 1.1. The edges of each node are sorted based on their priorities (obtained from scheduling heuristics) in a decreasing order. Therefore, the most left edge (first edge) is the one that the list scheduler would select at that point. That is why the first produced schedule by the DFS method is the one that the heuristic-driven list scheduler offers. The next produced schedules look almost the same with slight changes in the last instructions (last decisions made). While this approach makes a progress in exploring different instruction orders, it can take a long time to find a new and better schedule. As another method of searching, a randomised uniform search was introduced. This search is totally random and can produce *any* schedule in the solution space (all the possible schedules). The same applies to the BFS method without any randomisation. After implementing these methods for our problem, it has been observed the convergence to better schedules was very slow.

Another approach to randomise the list scheduler was introduced in [19]. The randomisation only took place as a tie breaker, i.e. when the scheduling heuristics consider the priorities of two instructions equal. As long as the scheduling heuristics choose an instruction, whether it is a good choice or not, the randomised iterative scheduler does not produce a different instruction order. This limits the scope of different schedules that are produced since it only affects some instructions.

The proposed solution in [6] is an optimisation based on meta-heuristics. It is a machine learning approach which tries to predict the best optimisation flags for compiling a set of applications. The improvement targets can be different such as: code size, execution time, and even compilation time.

1.3 Goals and Contributions

The goal of this research is reducing the length of schedules for a VLIW processor. The first step to achieve this, is making the scheduling pass iterative so that it could be run a user specified number of times (or trials) and select the shortest schedule over all

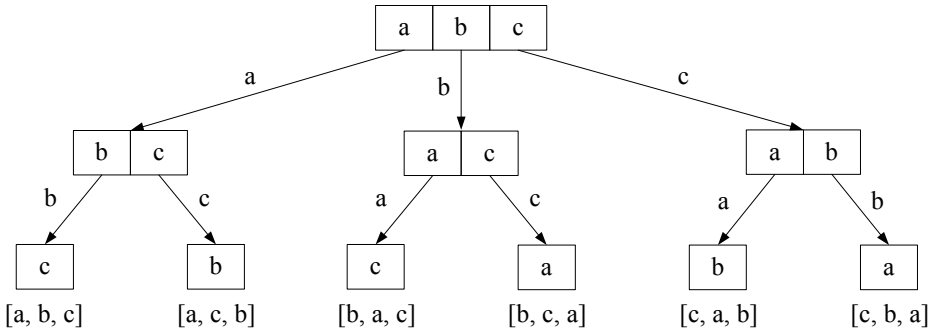


Figure 1.1: The decision tree for 3 independent instructions: a, b, and c.

trials as the final schedule. This is called the iterative scheduling throughout this thesis. The second goal involves finding better schedules in a reasonable amount of time. The solution space (all valid instructions order) can be huge and only a small portion of this space may contain good schedules. If the search strategy for finding better schedules does not use any intelligence, it can take a long time to find a good answer.

To address these concerns, the iterative scheduler proposed in this thesis iteratively explores schedules that are in the neighbourhood of the default schedule[2]. A directed search was performed in the solution space: whenever a neighbouring schedule with worse performance is reached, it will not be explored any further; otherwise the search continues in this direction. This corresponds to a simulated annealing with parameter $T = 0$. It has been observed that using a search like this converges faster to better schedules rather than a blind search. Unlike the approaches introduced in [10] and [19], our method does not involve changing the list scheduler to bring randomness to its decisions. Instead, the priority of instructions are randomly modified and the effects on the list scheduler are evaluated. This is illustrated in Figure 1.2. Our approach produces different schedules that are similar to the default schedule with a couple of instructions permuted. This helps us benefit from a fairly good solution (default schedule) while trying other schedules.

Another interesting problem that was tackled during this research was counting the number of possible topological orders for a Directed Acyclic Graph (DAG). The reason for doing so lies in the fact that *if the number of possible orders is less than the number of iterations to try different schedules*, then there is no point in randomly producing schedules while all of them can be tried.

Last but not least, the delay slot filling pass for EVP was slightly modified to adopt the new schedules better. A motivational example is studied in Section 4.3.1.

[2]The schedule that is obtained from the heuristic-driven list scheduler.

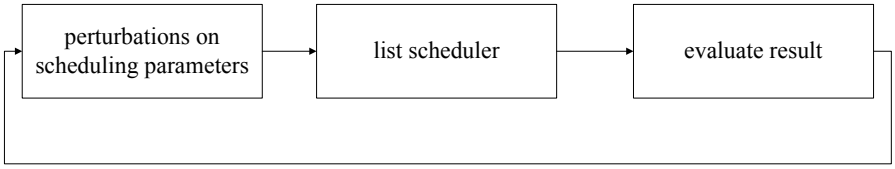


Figure 1.2: The 3 important steps in our iterative list scheduling algorithm.

To summarise, the contributions in this research are listed as follows:

1. A directed (local) search method was devised which starts from a fairly good solution (discussed in Section 3.2.1).
2. An algorithm was designed and implemented for getting out of local minima (discussed in Section 3.3).
3. An NP-complete problem was tackled to count the number of possible topological orders (discussed in Section 3.4).
4. The delay slot filler for EVP was modified to accommodate the changes in the schedules in a more efficient way (discussed in Section 4.3.1).

The rest of the thesis is laid out as follows:

- Chapter 2 - **Fundamentals:** It starts with introducing the GCC, especially its list scheduler. Then a motivational example is given to illustrate the possibility of better schedules beyond the scheduling heuristics. It ends with a brief description of the EVP processor.
- Chapter 3 - **The Iterative List Scheduler:** The main contributions are discussed in this chapter. It consists of 3 sections: the introduction of the directed local search algorithm (swapping); the *landing* as a way to get out of local minima; and how the counting of topological orders is implemented.
- Chapter 4 - **Iterative Framework:** It presents the implementation details about having a generic iterative scheduler in GCC.
- Chapter 5 - **Experimental Results:** The improvements for two sets of benchmarks are demonstrated in this chapter. It also represents the trade-off between the compilation time and the gained performance.

2 Fundamentals

This chapter begins with a brief explanation of the compilation process, followed by studying the list scheduling method used in GCC. Afterwards, an example is given to indicate that the scheduling heuristics might not always be the best solutions and still, a better scheduling order may be achieved. Finally, the target processor's architecture and features are briefly discussed.

2.1 Definitions

In this section, we define the terms that are widely used in this thesis:

1. **Instruction priority:** A number attached to every instruction in GCC which is used by the list scheduler (discussed in 2.3.2). The higher this number, the more important the instruction. GCC initialises this field with the critical path length which is explained in detail in Section 2.3.3.

2. **Basic block:** A straight line code sequence with no branches in except to the entry and no branches out except at the exit [8].

3. **Region:** A set of basic blocks grouped together based on the control flow graph. How regions are computed is not a concern in this study. However, for more information you may refer to [4].

2.2 Compilation

In this section a brief description of the main GCC compilation phases is given. More information regarding the internals of GCC can be found in [5]. As shown in Figure 2.1 the C/C++/Java tree obtained after parsing the input application, is first cleaned-up by language specific constructs resulting to a *Generic* tree. The name *Generic* comes from the fact that all input languages get converted to a common representation which further on allows generic optimisation phase. Afterwards, the nodes of the *Generic* tree are converted to a three-address representation obtained in this way, the *Gimple* tree. The *Gimple* trees are converted to the Static Single Assignment (SSA) representation [15, 3]. SSA allows efficient data flow analyses and optimising transformations (like vectorisation). The SSA Tree is converted back to a multiple assignment tree which gets linearised into a format called Register Transfer Language (RTL). This format is close to the processor Instruction Set Architecture (ISA) which makes it appropriate for target dependent optimisation. In fact, as shown by the right dotted arrow in compilation flow diagram, Figure 2.1, all target dependent passes operate on the RTL. In the initial stages, the RTL instructions are somewhat abstract, assuming an infinite number or

registers. However, during the later back-end phases they get transformed into real processor instructions operating on hardware registers and/or memory references [22].

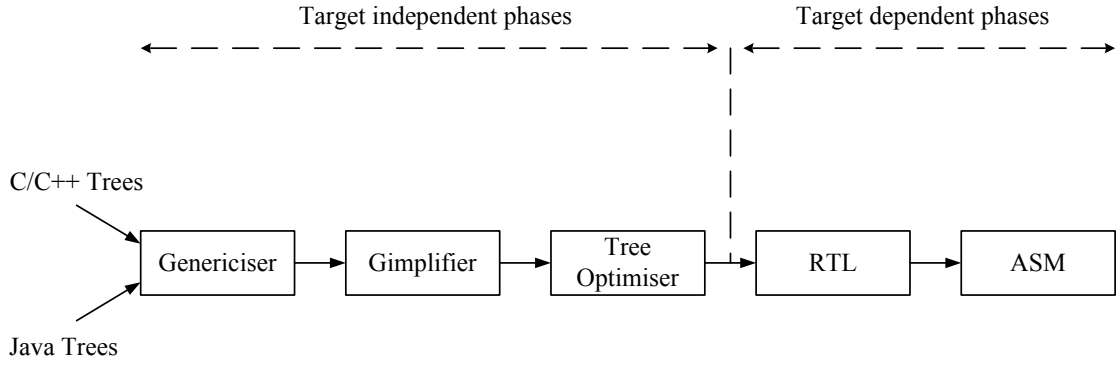


Figure 2.1: Compilation flow in GCC.

Around 200 passes are executed during the compilation. Each pass is responsible for performing a particular task such as Dead Code Elimination (DCE), instruction scheduling, Integrated Register Allocator (IRA), etc. The instruction scheduling pass occurs during the RTL phase. It is usually executed three times:

- **Sched1:** It happens before the register allocation. In this pass, an unlimited amount of registers are available to the scheduler and the dependency costs between the instructions are one clock cycle[1]. The purpose of this pass is to order the instructions, so that the register allocator [12] can work on that. The scope of the scheduler during this pass is a region of basic blocks (at least one). Sometimes instructions are moved from one block to another in the same region (code motion). At the end of this pass, there is no code duplication.

- **Sched2:** The second instruction scheduling pass occurs a few passes after the register allocation. Since there might be new instructions introduced, compared to *sched1*, because of likely memory spills, another scheduling pass is required. During this pass, the dependency costs represent the actual values and the instructions are mainly scheduled in the way they are supposed to be issued by the processor. The scheduling domain in this pass is one basic block.

- **Sched3:** The last scheduling pass is responsible for taking care of data and structural hazards between the basic blocks. It also fills the delay slots of branch instructions. This is done iteratively and can be time consuming. The scheduling scope is all the basic blocks belonging to the same function. In other words, during scheduling a block, other blocks might be considered and re-scheduled.

[1] Having dependency costs of one leads to decreasing the pseudo-register live ranges. Therefore, there will be less register pressure in the end. Nevertheless, one may apply the *-fsched1-accurate* flag to use the actual dependency costs in *sched1*.

These passes use the same list scheduler to organise the instructions. Since the scheduling order obtained at *Sched3* is the very same order printed in the assembly output, it is reasonably the first choice for randomised iterative scheduling. However, *Sched3* is already an iterative pass and it is not very practical to wrap it inside another iterative approach. This leaves us with *Sched2*, the classical basic block scheduler, as the first and main option for iterative scheduling. In this thesis, the feature of iterative scheduling is also added to *Sched1*, because it can have drastic influences on the register allocator. In Figure 2.2, the order of scheduling passes and the register allocation pass is depicted.

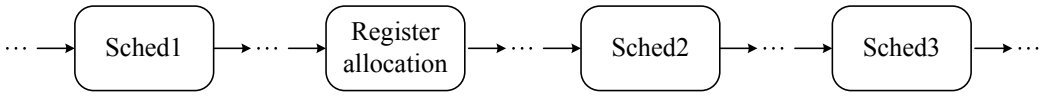


Figure 2.2: A few passes from the RTL phase.

2.3 Instruction Scheduler

Instruction scheduling is about reordering the instructions in such a way that the utilisation of the processor increases. This is achieved by extracting the ILP in the code and using it to cover the delay between producing and consuming instructions. While reordering the instructions, the functionality of the compiled program must be identical to the code written at the source level. The factor that the scheduler takes into account regarding this principle is the dependencies between instructions. Apart from this, the scheduler also guarantees that neither structural nor control hazards happen while the non-interlocked processor is issuing the instructions. The GCC applies a Deterministic Finite Automaton (DFA) based technique to avoid resource conflicts [14, 17].

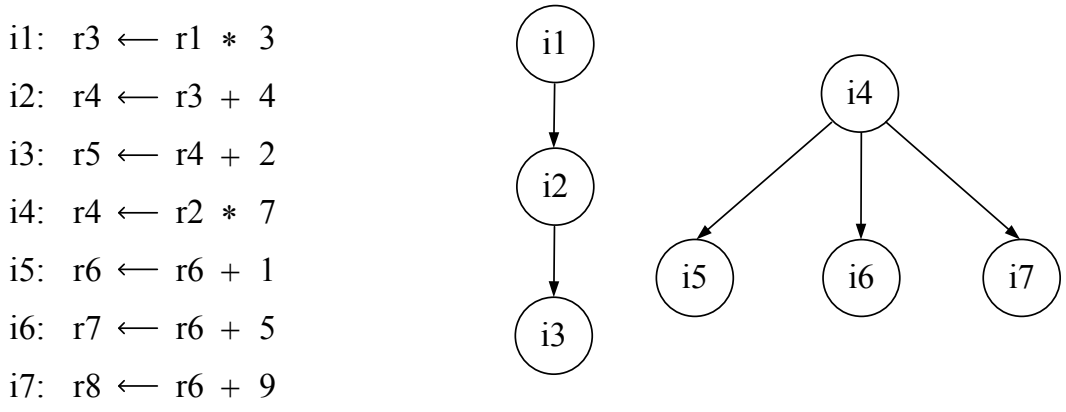
2.3.1 Data Dependence Graph

The data dependencies information can be represented by means of a DDG. Each node in this graph represents an instruction and each directed edge between a pair of nodes represents a dependency between them. The data dependence graph for a single basic block is a DAG. There are a few terms that are used frequently [20]:

- Successor: node v is said to be the *successor* or *immediate successor* of node u if there exists an edge (u, v).
- Predecessor: node u is said to be the *predecessor* or *immediate predecessor* of node v if there exists an edge (u, v).
- Descendants: all nodes that can be reached from a node.

- Source: a node without any incoming edge.
- Sink: a node without any outgoing edge.
- Edge weight: *weight* of the edge between two nodes is the dependency cost between the corresponding instructions.

Figure 2.3 depicts an instruction sequence and its DDG. Nodes $i1$ and $i4$ are source nodes; $i3$, $i5$, $i6$, and $i7$ are sink nodes. $i2$ is the predecessor of $i3$; and $i6$ is one of the successors of $i4$. The descendants of $i1$ consists of $i2$ and $i3$.



(a) Instruction sequence (b) Data dependence graph

Figure 2.3: An instruction sequence and its dependence graph.

2.3.2 Scheduling the Instructions

Now, let us have a closer look at how the scheduler operates. GCC applies a list scheduling algorithm to order the instructions. There are four queues maintained by the scheduler [22]:

- Scheduled: the list of scheduled instructions.
- Ready: the list of instructions whose data dependent predecessors are scheduled; also the required data latencies[2] are satisfied.
- Queued: the list of instructions that have all their predecessors scheduled but still are waiting for them to finish. It can also contain instructions which come from

[2]Data latency between instructions i and j is the minimum number of cycles that instruction j can be issued after i.

the ready list but are not scheduled for some reason (e.g. resource conflicts). Such delayed instructions are put back into the ready list in the next cycle.

- Pending: this list consists of instructions whose data dependent predecessors are not scheduled yet.

For an instruction to be inserted into the ready list, all of its predecessors must be scheduled. In the beginning of the scheduling procedure, the only instructions that qualify for this requirement are the sources in the DDG. At each time step, there can be more than one instruction available in the ready list. Making different decisions at this point leads to different scheduled sequences in the end. The ready list is a priority queue [2] and the instructions in this list are sorted by using the `rank_for_schedule()` function as the *comparator*. This function takes two instructions as input and after considering some scheduling heuristics, it determines which one of these two instructions is preferred over the other. The heuristics that are applied on normal instructions[3] are listed here in the order of importance:

1. Register pressure[4]: the instruction whose scheduling results in smaller register pressure is preferred.
2. Critical path length: this is explained in Section 2.3.3.
3. Number of successors: the one with more successors is considered more important.
4. Unique Instruction Identifier (UID): as the final tie breaker, the instruction with lower UID is preferred [5].

The ready list is sorted in a descending priority order. This means that the first element in the list is the most important one, the second element is the second most important instruction, and so on. To compare two instructions while sorting the ready list, these heuristics are used in the given order until there is a winner. So for example, if i and j are normal instructions which happen to have the same register pressure but different critical path lengths, then the corresponding heuristic is going to settle which one is preferred. Since most focus of the iterative scheduling is in *sched2*, it is safe to assume that the first heuristic being used is the critical path length. There are two unlisted heuristics related to *dispatch group scheduling* and *speculative instruction scheduling* [4]. Since they are not supported by EVP, the corresponding heuristics are

[3] "Normal instructions" refers to the instructions that are neither delay markers nor debugging instructions. The delay markers are used to fill the delay slots of the branches. Since it is desired to fill as many delay slots as possible, the scheduler is forced to choose them over any other option. The next most important instructions are the debugging ones and they have to be scheduled as soon as possible. Because these considerations are rather a set of rules than heuristics, they were not mentioned in the heuristics list.

[4] This heuristic is only used during *sched1* which is before register allocation pass. During *sched2* and *sched3* which occur after register allocation, there is no concept of register pressure.

[5] UIDs are assigned to the instructions in the same order that they appear in the source code. Therefore, an instruction with a lower UID occurs sooner at the source level than the one with higher value. This tie breaker wants to follow the same order.

never applied. It is worth mentioning that out of the four listed heuristics, the register pressure is a dynamic criterion while the rest are static and pre-calculated. Register pressure at a certain time step depends on how the instructions are scheduled until that time.

In Algorithm 2.1, the pseudo-code for scheduling is listed. This routine is repeated as long as the whole basic block is not scheduled. If there is nothing left for scheduling at any clock cycle, then the cycle advances and instructions are brought into the ready list. The CHOOSE-INSN() function is responsible for sorting the ready list and removing its first element. When an instruction is selected from the ready list, it must be ensured that there is enough resource available to issue it. This is verified by the CONFLICTS(). In case it is not possible to issue the instruction, it is queued in the pending list. After scheduling an instruction, some data dependencies are resolved. This can insert new instructions into the ready list in the next or even the current cycle if the cost of resolved dependency is zero. The UPDATE() function performs of this task. The key essence of ADVANCE-ONE-CYCLE() is about advancing the state of dependencies and resource usages to the next cycle.

Algorithm 2.1 List scheduler in GCC

Require: A basic block in the form of DDG
Ensure: Scheduled sequence of instructions

```
function SCHEDULE-BLOCK(basic-block)
    INIT(ready-list)                          ▷ Initialised with source nodes in DDG
    while not all instructions are scheduled do
        while ready-list ≠ [ ] do
            insn ← CHOOSE-INSN(ready-list)
            if CONFLICTS(insn)
                QUEUE(insn)                   ▷ Becomes ready again in next cycle
            else
                SCHEDULE-INSN(insn)
                UPDATE(ready-list)
            end if
        end while
        ADVANCE-ONE-CYCLE()
        UPDATE(ready-list)                    ▷ New instructions are added
    end while
end function
```

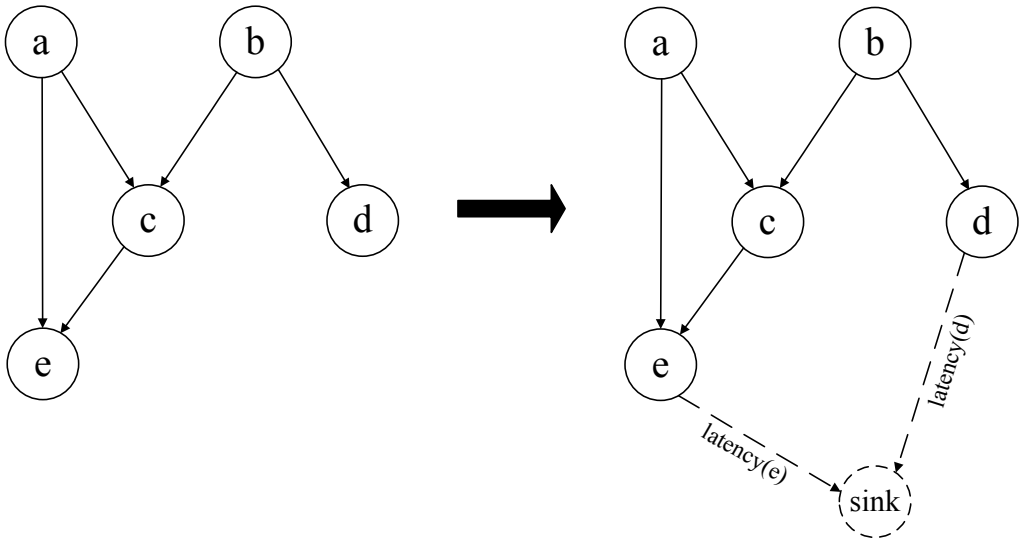
2.3.3 Critical Path Length

Critical path length is usually the first criterion for sorting the ready list. In this context, for the sake of brevity, we refer to the critical path length of an instruction as the *priority* of that instruction. *Priority* of an instruction is a weight assigned to it, representing the heaviest chain of dependencies starting from that instruction. If the instruction i is not the source of any dependency, its *priority* is the same as the number of clock cycles

it takes to execute it. But if there are some dependent instructions like j, for each of them let the sum of $priority_j$ and dependency cost between i and j be a candidate for $priority_i$. The *priority* of instruction i is the maximum of these values. This is illustrated in equation (2.1).

$$priority(i) = \begin{cases} latency(i) & |dependants(i)| = 0, \\ max(dependency\_cost(i,j) + priority(j)), & \text{otherwise.} \\ \quad \forall j \in dependants(i) & \end{cases} \tag{2.1}$$

To get a clear view of critical path length, assume a single node, called *sink*, is added to the graph. Moreover, new edges are introduced from any node in the graph with the output degree of 0 to this *sink* node. The weight of these edges is the latency of the source node. Critical path length for each node is the maximum weighted path from that node in DDG to the node *sink*. This is visually depicted for a sample DDG in Figure 2.4.



(a) A simple data dependence graph (b) Same graph with dummy *sink* node

Figure 2.4: Adding the *sink* node to the DDG.

For every instruction, which is a node in DDG, a flag is used to indicate whether the priority of that node is valid or not. This way the process of calculating the priorities for all nodes happens in $O(|N| + |E|)$ time. Algorithm 2.2 illustrates the `priority()` and `set_priorities()` functions pseudo-code in GCC.

Algorithm 2.2 The priority calculation in GCC

```
function PRIORITY(insn)
    if priority-status[insn] = VALID
        return priority[insn]
    end if

    if |dependants[insn]| = 0                    ▷ No instruction depends on insn
        priority[insn] ← INSN-COST(insn)
    else
        max-priority ← −1
        for all j ∈ dependants[insn] do
            this-priority ← DEPENDENCY-COST(insn, j) + PRIORITY(j)
            max-priority ← MAX(max-priority, this-priority)
        end for
        priority[insn] ← max-priority
    end if

    priority-status[insn] ← VALID
    return priority[insn]
end function

function SET-PRIORITIES(basic-block)
    for all insn ∈ basic-block do
        PRIORITY(insn)
    end for
end function
```

In Figure 2.5, an example is given that illustrates how *priorities* are assigned according to the equation 2.1. The number in each node represents its *priority* value. Since nodes c, e, and f have no children (dependent instructions), their *priority* is their latency. The weight of each edge is the dependency cost between the source and destination node. The edge (b, e) with cost 0 indicates that although b must execute before e, they may be issued in the same cycle. In other words, instruction e can be issued 0 (or more) cycle after b. This usually corresponds to an anti-dependency. Let us take a closer look at how the *priority* of node a is assigned to it. There are two dependent instructions on a. The dependency cost of (a, f) is 2 and the *priority* of f is 1, hence the regarding weight would be 3. On the other hand, the dependency cost of (a, d) plus the *priority* of d is $2 + 2 = 4$. The latter, being the higher value, is the priority of node a. In this DDG, a is the first instruction that will be scheduled based on the critical path length heuristic.

2.3.4 Motivational Example

Although the scheduling heuristics are empirically proven to lead to a fairly good result, the likelihood of a better solution should not be ignored. In this section, an example is reviewed which illustrates this fact.

Assume a machine capable of issuing at most one multiplication and two addition

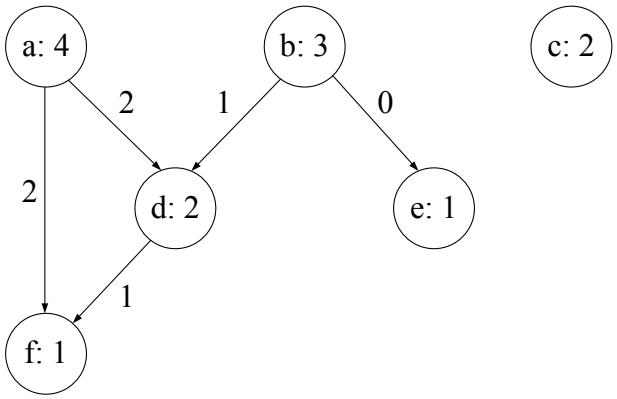
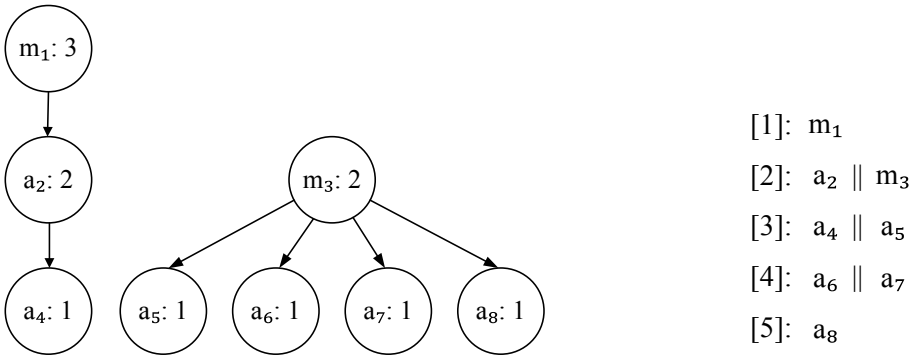


Figure 2.5: A DDG with critical path length assigned to each node.

operations in each cycle. The number of clock cycles needed to finish either of these operations (instruction latency) is one. Moreover, assume a basic block with a DDG as depicted in Figure 2.6a. Each m_i is a multiplication instruction and each a_i is an addition instruction. The subscript i represents the UID. For each node, the number after the colon is the *priority* (critical path length) assigned to that instruction. Nodes with higher values have more importance than those with lower values. In case the *priorities* of two instructions are the same, the one with lower UID is scheduled first.



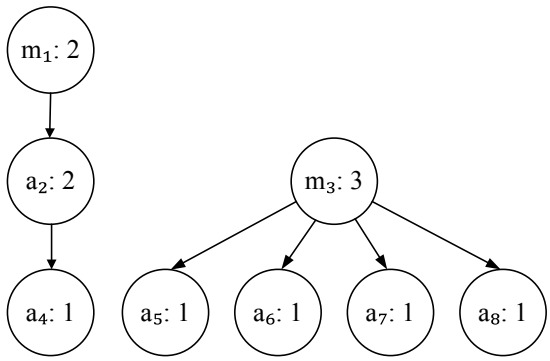
(a) Instructions with critical path priorities

(b) Corresponding schedule

Figure 2.6: A DDG and its schedule obtained by using critical path heuristic.

Let us see how the DDG in Figure 2.6a gets scheduled. At the first clock cycle, two instructions, m_1 and m_3, are ready. Because at most one multiplication can be issued in each cycle, the one with higher importance (m_1) is chosen. Since nothing else can be

scheduled in the first cycle, the clock advances and both m_3 and a_2 move to the ready list. Having one multiplication and two addition units available in each cycle makes it possible to issue both of these instructions in the second cycle. The rest of the instructions will be scheduled based on their UIDs because they have the same priority values. This leads to a scheduling order of $[m_1, a_2, m_3, a_4, a_5, a_6, a_7, a_8]$ with 53% utilisation[6].



(a) Instructions with arbitrary priorities

[1]: m_3

[2]: $m_1 \parallel a_5 \parallel a_6$

[3]: $a_2 \parallel a_7$

[4]: $a_4 \parallel a_8$

(b) Corresponding schedule

Figure 2.7: Same DDG with another priorities ended up in a better schedule.

In Figure 2.7a, the priority of m_1 and m_3 are exchanged and the priority of the rest is left intact. While scheduling this DDG, m_3 is preferred over m_1 at the first clock cycle. In the next cycle, three instructions are ready: m_1, a_5, and a_6. According to the resource constraints, all of these three instructions can be issued resulting in a 100% utilisation. The final scheduled order is $[m_3, m_1, a_5, a_6, a_2, a_7, a_4, a_8]$ in this way, resulting in 66% utilisation. This order is one cycle shorter than the previous, as seen in Figure 2.7b.

It is worth mentioning, the advantage of the 4 cycle schedule over the 5 cycle one is because of selecting instruction m_3 first. Since the approach introduced in [19] only takes over the scheduler decisions when there is a tie, it never finds the shorter schedule in this example. This is because the priority of m_1 and m_3 are different. One can look at this as the bad choices of scheduling heuristics propagates in all schedules.

This example shows that although applying scheduling heuristics may lead to a good instruction sequence, but there can be other ones which are better. To find such possible solutions, the scheduler must go beyond these heuristics to come up with different scheduling orders.

2.4 The Embedded Vector Processor

EVP was initially developed by NXP Semiconductors. EVP is mainly used in digital signal processing domain. At the time of writing, it is a product of ST-Ericsson and is

[6]This number is the average of the machine's utilisation in each cycle. For instance, in the first cycle only the multiplication unit is busy and the two addition units are idle. Therefore, the utilisation is 33%. Following the same concept, the utilisation is 66%, 66%, 66%, and 33% in cycles 2 to 5 respectively.

being developed for wireless and mobile markets to handle 2G, 3G and 4G standards. These standards mainly require a lot of data stream processing. Hence, the design of this processor supports multiple operations on 256 bit size vectors [21]. Most of the algorithms in this domain benefits from parallelisms in their computations. The compiler is the sole factor in exposing the ILP to the processor, because EVP is designed in such a way to have a simple and power efficient architecture.

In the forthcoming sections, the architecture of EVP is presented and a few facts that are related to the scope of this thesis are illustrated.

2.4.1 The Architecture

EVP is a Very Long Instruction Word (VLIW) Digital Signal Processor (DSP) with vector processing capabilities. It supports parallel execution of multiple scalar and vector operations. The processor core block diagram is depicted in figure 2.8. Each of those units has access to its local register files and also limited access to those of the neighbour's register files. In most of the cases, data-dependant operations can be scheduled on the consecutive cycles due to the pipelined execution and bypass network. One EVP instruction can be composed of up to 13 operations. Each of these operations specifies the task each functional units must perform [22].

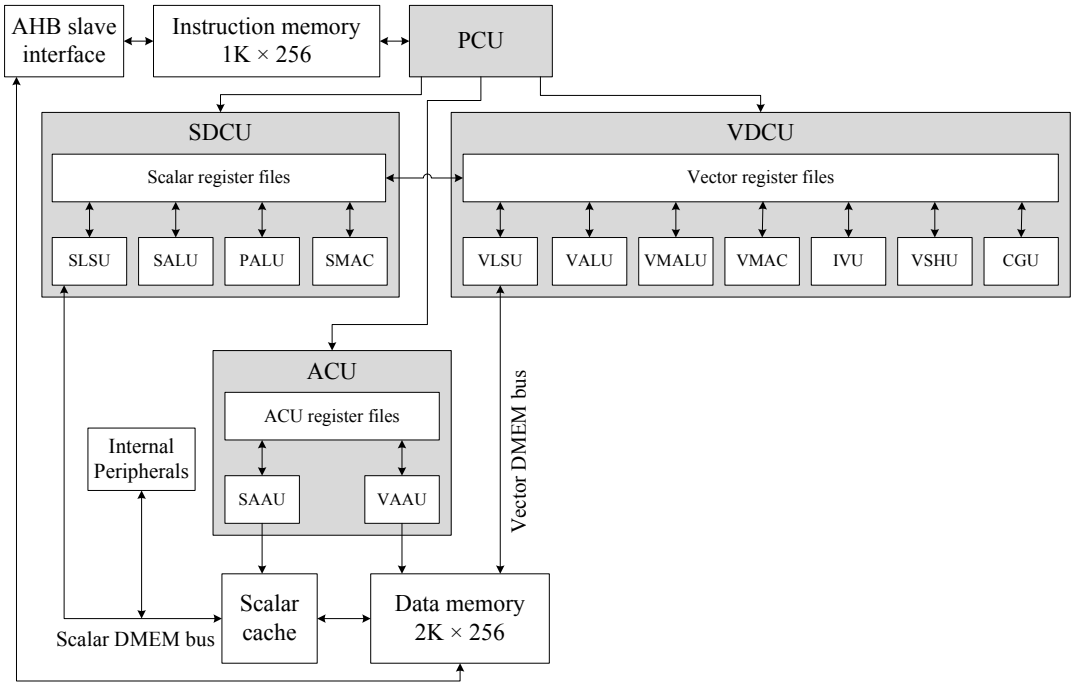


Figure 2.8: EVP core block diagram.

The EVP core is composed of four main units, each of which includes a number of

registers and functional units:

1. **Program Control Unit:** The PCU fetches instructions from the program memory, decodes them, and controls the other main units. It also handles the hardware loops through its special purpose registers.

2. **Scalar Data Computation Unit:** The SDCU contains the register files and functional units for handling scalar data. The Scalar Load/Store Unit (SLSU) is responsible for performing load and stores operations for both scalar and predicate registers. SALU is a Scalar Arithmetic Logic Unit, while PALU is a Predicated Arithmetic Logic Unit that takes care of 1-bit predicate registers. The last unit, Scalar Multiply/Accumulate (SMAC), covers both multiplication and multiply-accumulate operations [21].

3. **Vector Data Computation Unit:** The VDCU is made of multiple vector register files and functional units for handling vector operations. The Vector Load/Store Unit (VLSU) handles load and stores operations for both regular and mask vectors. To perform arithmetic/logical operations on vector and vector mask registers, it uses Vector Arithmetic Logic Unit (VALU) and Vector Mask Arithmetic Logic Unit (VMALU) respectively. The Vector Multiply/Accumulate (VMAC) handles multiplication with optional accumulation on vectors. The Intra Vector Unit (IVU) makes it possible to perform tasks on a single vector (e.g. finding the maximum of elements in one vector). The Vector Shuffle Unit (VSHU) shuffles the elements of a vector according to a pattern. The last functional unit is Code Generation Unit (CGU) which generates an application specific code that must be inserted into the instruction streams (for example in the Universal Mobile Telecommunications System (UMTS) standard that is a part of 3G, scrambling the code is needed [9]) [21].

4. **Address Computation Unit:** The ACU calculates the addresses needed in scalar or vector operations (addressing data memory accesses).

In principle all scalar and vector units can be operated in parallel in one instruction. For various reasons, the VLSU and SLSU units operate on the same data memory. In order to support these two simultaneous memory operations per cycle, a system with caching is used. To make this happen, the VLSU is connected directly to the data memory, while the SLSU is connected to the data memory through a scalar cache.

2.4.2 Features

A few properties in the EVP must be considered before going into more detail on the iterative scheduling:

Exposed non-interlocked pipeline: EVP cannot stall its multi-stage pipeline, so the scheduler must deal with that and insert NOPs whenever needed.

In-flight Scheduling: In-flight scheduling takes advantage of the fact that some instructions have an execution latency of multiple cycles. The destination register, where the result of such an operation is written to, can be used for other purposes while the operation is in progress. A sample code using this feature is represented in Figure 2.9.

```
load r0, 0x1000     /* [1] instruction latency of "load" is 3 clock cycles */
add  r0, r1, r2     /* [2] instruction latency of "add"  is 1 clock cycle  */
mov  r2, r0         /* [3] r0 is the result from  "add"                    */
sub  r1, r1, r0     /* [4] r0 is the result from  "load"                   */
```

Figure 2.9: An example of in-flight scheduling.

Packed Instruction Words: The instructions that are supposed to be issued at the same clock cycle are packed together in a VLIW format. There must be no resource conflict neither amongst any of these instructions nor between the previous issued ones which are still in the pipeline. The maximum opcode size for a VLIW is 24 bytes and at most 13 instructions can be packed together. It must be noted that the functional unit assigned to each instruction is decided by the `best_alternative()` function in the compiler. It applies a greedy approach to come up with a functional unit for an instruction in such a way that maximum number of instructions can be packed together in the current VLIW. In Figure 2.10, the assembly equivalent of VLIWs is illustrated. The numbers in the square brackets are the clock cycles, the `||` (parallel) operator indicates that the corresponding instructions are packed together. The comment at the end of each line represents the used functional unit and the UID of the instruction.

```
                                          /*  FU    UID     */
// [1]
       vmove8              vr5, vr4       // VALU insn_id:16
||     vmove_vlsu8         vr4, vr1       // VLSU insn_id:17
||     vmove_vshu8         vr1, vr2       // VSHU insn_id:18

// [2]
       vmuli16             vr2, vr3, vr0  // VMAC insn_id:22
||     vload_post_update   vr3, ptr0, 32  // VLSU insn_id:20
||     vadd16              vr1, vr1, vr0  // VALU insn_id:23

// [3]
       vadd16              vr4, vr4, vr0  // VALU insn_id:24
||     vstore_post_update  ptr8, 32, vr5  // VLSU insn_id:25
```

Figure 2.10: Assembly output of the compiler.

Branch Delay Slots: The branching mechanism in EVP is based on visible branch delay slots. This means that a number of instructions following a branch are always executed. Depending on the branch type, the number of delay slots varies from 5 to 7 clock cycles.

Hardware Loops: By means of its dedicated registers stack, EVP is able to handle up to 3 nested hardware loops. The required information for using this feature is the starting and ending addresses of the loop body, along with the number of iterations. The loop counter is either an immediate value, which is known during the compilation time, or specified during the runtime[7]. The compiler detects potential hardware loops in the program and transforms each of them into a single hardware loop instruction.

[7]The number of delay slots for the immediate and dynamic version is 5 and 7 clock cycles respectively.

3 Iterative List Scheduler

The GCC compiler takes as input the source code of a program and applies numerous passes of transformations to it before translating it into machine code. The instruction scheduling pass plays an important role in maximising the target machine's utilisation, hence achieving faster applications. Since the scheduling is well known to be an NP-complete problem, the list scheduler in GCC applies heuristics to produce near optimal schedules. While these heuristics yield to a fairly good solution (with short compilation time), the likelihood of having a better schedule cannot be ignored.

The aim of this study was to improve the performance of the static instruction schedule in GCC's list scheduler. We propose an iterative scheduling pass that randomly produces schedules other than the default one[1] and picks the best as the solution. The number of iteration is user-specified and is read from the command-line. While reducing the cycle counts is the key goal of this project, the improvements should be in a reasonable amount of time. To accomplish this, a directed search towards better schedules is applied. Our approach benefits from starting with the default schedule which is a fairly good solution. This initial solution is changed each iteration to obtain better schedules. Another problem that was taken into account was the possibility of being trapped in a local minimum and a solution to that is discussed. Last but not least, a method is introduced to calculate the number of topological orders in a DAG. This number represents the size of the *solution space* and helps us to determine if the number of iterations is big enough to cover entire space. If so, all of the solutions are produced instead of trying them randomly.

3.1 Terminology and Definitions

Before delving into more details, we must clarify the terminology that is frequently used in describing our methods.

- **Scheduling parameters:** These are the priorities of instructions in the basic block. They are initialised with the Critical Path Lengths (CPLs) of the corresponding instruction.
- **Schedule:** This refers to an instruction order which is produced by the list scheduler.
- **Current schedule:** Our approach involves scheduling a basic block iteratively. In each iteration, the basic block is scheduled once. At any iteration the corresponding result of the scheduler is referred to as the *current schedule*.

[1]The default schedule is the one that the list scheduler constructs using the scheduling heuristics discussed in Chapter 2.

- **Default schedule:** As explained in the previous chapter, the list scheduler applies a few scheduling heuristics so as to order the instructions. The schedule obtained from using these heuristics are referred to as the *default schedule*. Later we will see that we modify the scheduling parameters to get different schedules as opposed to the default schedule.

- **Neighbour schedule:** The current schedule depends on corresponding parameters at that iteration. When we swap two elements of these parameters (more detail in Section 3.2.1), the next schedule based on these new parameters is called the *neighbour* of the current schedule. In Figure 3.2b, possible schedules for a DDG are depicted. Every two schedules (nodes) that are connected to each other, are considered neighbours.

- **Performance:** It is the number of cycle count for the scheduled basic block to execute.

3.2 Making the Scheduling Pass Iterative

As discussed in Section 2.3.2, the list scheduler uses a few scheduling heuristics to construct a solution (schedule). These heuristics are deterministic. In other words, given the same input set (the DDG), the heuristics assign the same priorities to the instructions and the scheduler takes the same decisions. Therefore, in order to get different schedules on the same input, our approach modifies the priority of instructions. After the modification, we let the list scheduler work based on these new priorities to schedule the basic block. The achieved cycle count from this scheduling is then evaluated to decide what to do next. This is detailed in Section 3.2.1.

More formally the *priority vector* $P = \langle p_1, p_2, ..., p_n \rangle$, where p_i is the priority of $insn_i$[2], is used as the parameter that the iterative scheduler modifies. The dimension of this vector, n, is the number of instructions in the basic block. One can consider the priority vector as a point in an n dimensional space. The list scheduler then becomes a function that maps every vector in this domain to one and only one image in the range of valid schedules. With this point of view, instead of looking for different (and better) schedules explicitly, we search in this n dimensional space of the priority vector to obtain a good schedule.

The changes on the priority vector are made randomly before the scheduling starts. The general idea involves making random changes, invoking the scheduler, and evaluating the performance. These steps are executed in a loop for `number-of-iterations` times which is specified in compilation command-line[3]. Algorithm 3.1 demonstrates this idea. Two methods were implemented to change the parameters (priorities): *random swapping* and *random landing*. These methods are discussed in detail in next sections.

[2]Instructions are enumerated according to a key, i.e. their UIDs.
[3]The compilation flags for iterative scheduling is explained in Section 4.4.

Algorithm 3.1 Main concept of the iterative scheduler

Require: A basic block in the form of DDG; instruction priorities
Ensure: Scheduled sequence of instructions

```
function ITERATIVE-SCHEDULE(basic-block)
    for i = 1 → number-of-iterations do
        insn-priorities ← CHANGE-SCHED-PARAMETERS(basic-block)
        SCHEDULE(basic-block, insn-priorities)
        EVALUATE-SCHEDULE(basic-block)
    end for
end function
```

3.2.1 Random Swapping of Priorities

The swapping method is the key strategy in this research. It partially modifies the scheduling parameters during each iteration, in order to try a different schedule than the current one. By means of this method, we transform the current schedule to another that is slightly different. Our method benefits from a directed search that is detailed in this section.

The swapping method consists of selecting two instructions randomly and exchanging their priorities. When these two instructions appear in the ready list together, the order in which the list scheduler would select them is reversed. In Algorithm 3.2, the steps of iterative scheduling with random swapping are presented. First, the basic block is scheduled using the default scheduling heuristics. The performance of the default schedule is set as the reference (`best-performance` variable) that must be improved. The `priorities` variable is the priority vector initialised with the CPL of instructions. Next, the iterative scheduling begins. In each iteration, two instructions are picked randomly (detailed in Section 3.2.2 as Algorithm 3.3) using Pseudo-Random Number Generator (PRNG) and their priorities are exchanged. This is followed by scheduling the basic block using this modified priority vector and evaluating the performance. If the performance improves, the priority vector is stored as the solution to the problem and the performance reference is updated. If the performance degrades, the last swap is undone. In case the performance does not degrade, the modification on priority vector is kept, so that the next iteration modifies this vector. Next section demonstrates an example of this algorithm to make it clear, but meanwhile we study its properties.

The main idea behind devising the swapping method was changing the scheduling parameters slightly. The resulting alteration in the schedule is a transition from the current schedule to one of its neighbours in the solution space. Moreover, there is another feature to this method; it is possible to start the search from the part of solution space that is likely to have better schedules. This was achieved by initialising the priority vector with the CPL of instructions. As it has been already established, we want to improve upon the schedules which are obtained by using the scheduling heuristics. While there may be room for improvement for schedules produced by using these heuristics, it has been empirically proven that these schedules are fairly good. Hence, we use them as our starting point.

The swapping strategy utilises a *directed search*: whenever a neighbouring schedule with a worse performance is reached, it is not going to be explored any further. In other words, if a swap is done on the priority vector that led to a schedule with larger cycle count, then the iterative scheduler goes back to the previous priority vector to try another swap; otherwise, it continues to search in this direction (this new priority vector will be used in next iteration). This corresponds to a simulated annealing with parameter $T = 0$ [18]. It was observed that using a greedy search like this converged faster to better schedules rather than a blind search which is totally random.

In swapping method, the list scheduler can still benefit from the scheduling heuristics. We already established that the priorities of instructions are initialised with CPLs. As explained in Chapter 2, when two instructions have the same priorities, the list scheduler in GCC uses other heuristics to decide which one should be scheduled first. Therefore, although we are modifying the priorities of instructions by swapping them, there can be instructions in the ready list with the same priorities that need applying other scheduling heuristics. Which is why, our method is a combination of a modified CPL heuristic and other unmodified scheduling heuristics.

After discussing about the swapping and its properties, we present an example in next sextion to show how it actually works.

Algorithm 3.2 Directed search swapping used in iterative scheduler

```
function ITERATIVE-SCHEDULE(basic-block)
    priorities ← CRITICAL-PATH-LENGTH(basic-block)
    SCHEDULE(basic-block, priorities)          ▷ No parameter is modified at this point
    best-performance ← CYCLE-COUNT(basic-block)
    best-priorities ← priorities

    for i = 1 → number-of-iterations do
        (i, j) ← PICK-TWO-INSN-RANDOMLY(basic-block)
        priority[i, j] ↔ priority[j, i]    ▷ Priorities of instructions i and j are swapped
        SCHEDULE(basic-block, priorities)
        performance ← CYCLE-COUNT(basic-block)

        if performance < best-performance
            best-performance ← performance
            best-priorities ← priorities
        else if performance > best-performance
            priorities[i, j] ↔ priorities[j, i]                ▷ Undo the last change
        end if
    end for
end function
```

In Table 3.1, an example is provided to illustrate the behaviour of the random swapping with directed search policy. There are 5 instructions involved: i_1, i_2, i_3, i_4, and i_5. Initially (at iteration 0), these instructions are scheduled with their actual CPL values and a performance of 7 clock cycles is obtained. The first iteration works on the original priority vector and swaps its first and fifth elements. The second iteration continues by

working on the modified priority vector obtained from the previous iteration and after changing it again, the result of scheduling shows improvement (7 cycles → 6 cycles). In the forth iteration, the modification made on the priority vector brings degradation (6 cycles → 7 cycles). This is why in the fifth iteration, the last modification is ignored and the new swap is taking place on the previous priority vector: $\langle 1, 3, 4, 2, 3 \rangle$.

Table 3.1: Swapping strategy in action

Iteration	Priority vector	Swap	Priority vector after swap	Cycle
0	$\langle 3, 2, 1, 3, 4 \rangle$	-	$\langle 3, 2, 1, 3, 4 \rangle$	7
1	$\langle 3, 2, 1, 3, 4 \rangle$	(i_1, i_5)	$\langle \mathit{4}, 2, 1, 3, \mathit{3} \rangle$	7
2	$\langle 4, 2, 1, 3, 3 \rangle$	(i_1, i_3)	$\langle \mathit{1}, 2, \mathit{4}, 3, 3 \rangle$	6
3	$\langle 1, 2, 4, 3, 3 \rangle$	(i_2, i_4)	$\langle 1, \mathit{3}, 4, \mathit{2}, 3 \rangle$	6
4	$\langle 1, 3, 4, 2, 3 \rangle$	(i_4, i_5)	$\langle 1, 3, 4, \mathit{3}, \mathit{2} \rangle$	7
5	$\langle 1, 3, 4, 2, 3 \rangle$	(i_2, i_3)	$\langle 1, \mathit{4}, \mathit{3}, 2, 3 \rangle$	6
6	$\langle 1, 4, 3, 2, 3 \rangle$	(i_1, i_5)	$\langle \mathit{3}, 4, 3, 2, \mathit{1} \rangle$	5

3.2.2 Selection of Two Instructions for Swapping

Although, any pair of instructions can be randomly selected for swapping, we reduce the number of choices to pairs that are likely to make a change. This reduction of choices is performed by a filter which renders some pairs useless for swapping. Three main criteria were used to construct the filter:

1. The priority of the two selected instructions must not be equal; otherwise, swapping them makes no difference.

2. The chosen pair must not have been chosen in previous iteration; otherwise, swapping them will undo the last change.

3. For the selected pair (i, j), there must be no path in the DDG from i to j or vice versa (one should not be the ancestor/successor of the other).

As it was established earlier, swapping the priority of two instructions is considered as exchanging their importance between *each other*. Nevertheless, if the priorities of a parent and a child node in a DDG are exchanged, the parent still appears before the child in the schedule. This is because an instruction gets ready to be scheduled only after all of its predecessors are scheduled. To get a better understanding of this criterion, we discuss the example in Figure 3.1 first. In Figure 3.1a, according to the DDG, instruction a must execute before b. In Figure 3.1b, six[4] possible permutations of a priority vector for 3 instructions are listed. The first element in this vector represents the priority of a; the second and third elements represent the priority of b and c respectively. During the scheduling, the instruction with higher priority value is scheduled first. This is how the corresponding schedules are produced.

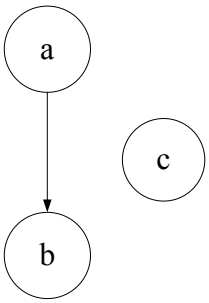
[4]Number of different permutations for three elements is $3! = 6$

Algorithm 3.3 lists the PICK-TWO-INSN-RANDOMLY() which is used in Algorithm 3.2. This algorithm applies the three mentioned criteria to select two instructions. In special cases when nothing can be returned from this function, e.g. when the priority of all instructions are the same, the iterative scheduling stops.

Algorithm 3.3 Selecting two instructions to swap their priorities

Require: Instructions of the basic block; the last selected pair
Ensure: Two instructions

```
function PICK-TWO-INSN-RANDOMLY(basic-block)
    repeat
        i ← a random instruction in basic-block
        j ← a random instruction in basic-block
        if priority[i] = priority[j]
            continue
        end if
        if (i, j) = last-pair                    ▷ last-pair is a global variable
            continue
        end if
        if j ∈ successors(i) or i ∈ successors(j)
            continue
        end if
    until 0
    last-pair ← (i, j)
     return (i, j)
end function
```



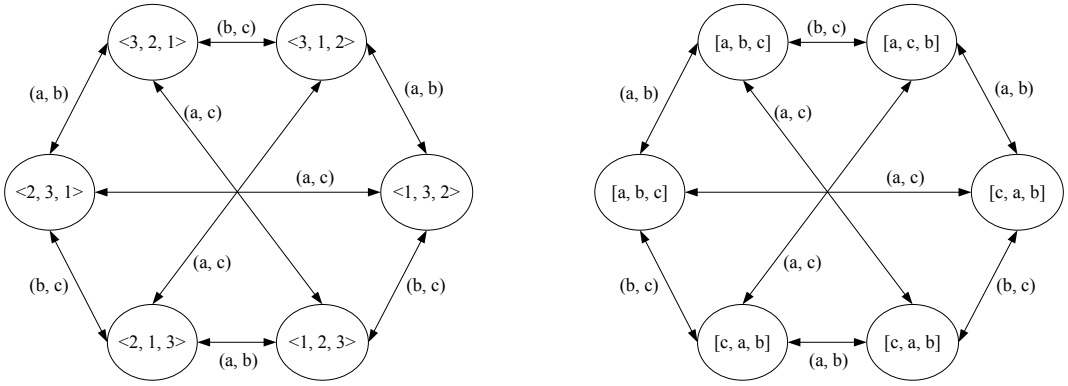
a) A sample DDG

#	Priority vector for instructions: a, b, c	Corresponding schedule
1	< 1, 2, 3 >	[c, a, b]
2	< 1, 3, 2 >	[c, a, b]
3	< 2, 1, 3 >	[c, a, b]
4	< 2, 3, 1 >	[a, b, c]
5	< 3, 1, 2 >	[a, c, b]
6	< 3, 2, 1 >	[a, b, c]

b) Priority vectors and their schedules for this DDG

Figure 3.1: All possible permutations of a priority vector for a DDG with 3 instructions.

In Figure 3.2a, all priority vectors for the example in Figure 3.1 are listed. There are 3 swaps available for each. Edge labels indicate that a swap is performed on the corresponding elements of the vector. For example, (a, c) means that the first and third elements of the vector are exchanged. In Figure 3.2b, the priority vectors are replaced with their corresponding schedules as listed in Figure 3.1b. In this small example it can be observed how swapping priorities affects the schedules.



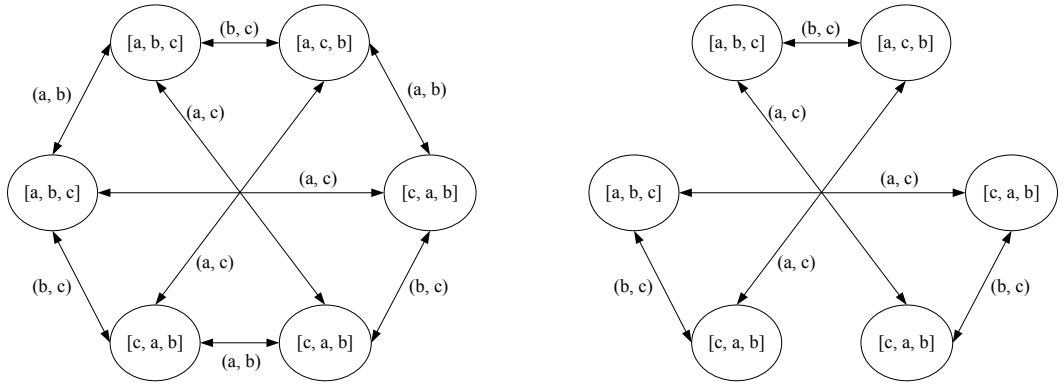
a) Priority vectors with one swap transitions

b) Priority vectors are replaced with corresponding schedules

Figure 3.2: Transitions between priority-vectors/schedules with one swap.

According to the three filtering criteria discussed in the beginning of this section, we reduce the number of pairs that can be chosen for swapping. This affects the new schedules that we can get from the current one. Figure 3.3 contrasts the difference between swaps with and without filtered choices. According to the third filtering criterion and the DDG depicted in Figure 3.1a, the priorities of a and b should not be exchanged. Figure 3.3b depicts the scheduling transitions diagram without the (a, b) edges. It is obvious at the first glance that two of these edges were actually useless, because they made a transition from one schedule to another which were the same ($[a, b, c] \leftrightarrow [a, b, c]$ and $[c, a, b] \leftrightarrow [c, a, b]$). The right (a, b) edge in Figure 3.3a that was removed is a transition from a schedule to another which is different ($[a, c, b] \leftrightarrow [c, a, b]$). However, it is still possible to transform either of these schedules to another by performing three swaps.

The filtering introduced in this section is an attempt to avoid useless choices. It is implemented in such a way that it does not eliminate the chances of producing different schedules. However, some extra swaps might be needed to transform one schedule to another.



a) Possible swaps at each state before applying the filter

b) Possible swaps at each state after applying the filter: (a, b) edges are removed

Figure 3.3: Unfiltered vs. filtered swaps and the effect on the transitions of the schedules.

3.3 Random Landing

Because of the directed search, repeatedly applying swaps over each other as introduced in the previous section might be prone to being trapped in a local minimum in the solution space. In such situations, it is useful to escape the minimum by generating a fresh, randomly produced schedule and performing a local search (swapping) from there.

Before we introduce our approach, we must define *landing options*. The landing options for a list are the possible ways that we can insert a new element into this list. For example, if there is a list such as $S = [a, b, c]$, the landing options for inserting a new element are: before a, before b, before c, and after c. These options are notated by "_" in the list $[\_, a, \_, b, \_, c, \_]$.

Our random landing approach consists of the following steps:

1. Let U be a set of instructions. Initially U contains all instructions in the basic block: $U = \{i : i \in BB\}$.

2. Let S be an ordered list of instructions. Initially S is empty: $S = [\,]$.

3. Until U is empty, the following steps are repeated:

   (a) $insn = remove\_element(U)$; The order does not matter.

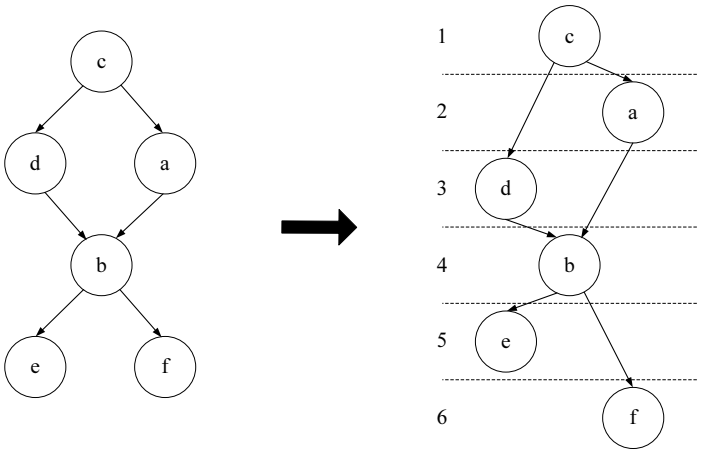
   (b) Let V be the set of all *valid landing options* in S where the insertion of $insn$ respects the order imposed by dependencies[5]. Randomly select an element v_i from V, and insert $insn$ into S at the landing indicated by v_i.

These steps are listed in Algorithm 3.4.

Algorithm 3.4 Landing instructions to produce a total random order

Require: basic block in the form of DDG
Ensure: A random order of instructions that respects DDG

function LANDING(*basic-block*)
  $U \leftarrow$ all insn $\in$ *basic-block*
  $S \leftarrow [\,]$
  while $U \neq \emptyset$ **do**
    $i \leftarrow$ an insn $\in U$ ▷ The order that elements are picked is not important
    $U \leftarrow U - \{i\}$
    $V \leftarrow$ VALID-LANDINGS(i, S) ▷ After ancestors and before successors of i that are in S
    $v \leftarrow$ one random landing option $\in V$
    INSERT(i, S, v)) ▷ Insert i in S at landing option v
  end while
  return S
end function



a) An example DDG b) A valid order

Figure 3.4: Producing an order while respecting the precedence.

In what follows we exemplify a random instruction order produced by random landing. The DDG of this example is depicted in Figure 3.4a. Nodes are picked from the U set in an arbitrary order: a, b, c, d, e, f. Since the list S is empty, there is only one position at which to insert a. After the insertion, $U = \{b, c, d, e, f\}$ and $S = [a]$. Next, b is removed from U to be inserted into S. According to the DDG, b must occur after a, so it must appear after a in the partial list S. Next, while inserting c into S, the

[5] *insn* can only be inserted into landings that are after any ancestor of *insn* that exists in S. *insn* also must be inserted before any of its successors that might be in the S.

DDG specifies that it must occur before a. Therefore, c must be added before a in list S. By doing this we have $U = \{d, e, f, g\}$ and $S = [c, a, b]$. For node d, the situation is different. It can be inferred from the DDG that d must occur between c and b. While respecting this dependency, there are two options: inserting d either between c and a, or between a and b. One of these options is chosen randomly (in this example the second one is selected, i.e. between a and b). This leads to $U = \{e, f\}$ and $S = [c, a, d, b]$. The only valid place to insert e into S is as the last element, after b. Finally, f must be injected after b. This leads to two choices: before e or after e. Again, the decision is made randomly and is decided to insert f after e. The final order is $S = [c, a, d, b, e, f]$ as depicted in Figure 3.4b. Since $U = \{\}$ at this point, the algorithm is finished. This way, after a series of random choices, an order is obtained which respects the DDG and has a higher chance of never having been tried. This order can be imposed on the scheduler by a priority vector like $P = \langle 5, 3, 6, 4, 2, 1\rangle$ where the nodes are enumerated alphabetically (5 is the priority of a, 3 is the priority of b, and so on.)

In each iteration, the swapping strategy uses the Pseudo-Random Number Generator (PRNG) once to change only two elements of the priority vector. On the other hand, the random landing method relies on consecutive use of the PRNG to construct a new priority vector. This increases the likelihood of producing a new priority vector that the iterative scheduler never worked on. If the DDG was not considered while (randomly) constructing the priority vector, the probability of producing different schedules would not be equal. To get a better view of this, look at Figure 3.1b. If a uniform PRNG is used, the probability of producing each vector is equal, whereas the probability of producing the 3 possible schedules are not (for instance, $[c, a, b]$ happens half of the times). However, the random landing approach produces only 3 distinct priority vectors: ($\langle 2, 1, 3\rangle$, $\langle 3, 1, 2\rangle$, and $\langle 3, 2, 1\rangle$). Each of these vectors lead to a unique schedule. Hence, the probability of producing each of schedules is equal.

3.4 Counting the Topological Orders

There is no need to search the solution space randomly in case that the given number of iterations[6] is enough to produce *all* valid schedules. There are two disadvantages if the random swapping approach is used in this scenario: first, there is a chance that the whole solution space might not be covered because of possible duplications; second, the iterative scheduler may produce all valid schedules randomly in early iterations and spend the rest of iterations producing duplicate schedules, which is the result of not knowing if the whole solution space is covered. However, to select the brute-force strategy in this scenario, we need to know how many different schedule exist for current basic block. If the iterative scheduler knows that the number of all schedules is going to be less than the given number of iterations, then it produces all schedules one after another during each iteration. To know the number of schedules we have to count the number of all topological orders which is known to be an NP-complete problem.

In [11], an approach is introduced to count the number of topological orders. It involves partitioning the graph into independent static vertices; in case there is only one

[6]It is a user-specified number that is read from a compilation flag.

static vertex in the graph which covers the whole graph, a brute-force enumeration is used to find the number of topological orders. However, the method we propose relies on transformations and makes no use of brute-force.

3.4.1 Preliminary

A topological order of a Directed Acyclic Graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge e_{ij}, then u appears before v in the ordering. A topological order of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right [2]. Let E be the set of all edges in the DAG. Each edge e_{ij} imposes the order: i occurs before j or more briefly, $i \rightarrow j$. Some edges may be redundant because the order they impose is already implied by other edges. For instance, if there are edges $i \rightarrow j$, $j \rightarrow k$, and $i \rightarrow k$, the last one can be removed because the order imposed by it is implicitly satisfied by the presence of the first two edges. Since the removal of redundant edges does not affect the number of topological orders possible, the *counter* works on DAGs where redundant edges had been removed.

Definition 3.1 *Let G be a DAG, and T_G the set of all topological orders of G; we define $count(G) = |T_G|$*

Definition 3.2 *A node with an incoming degree of zero is a* source

Definition 3.3 *A node with an outgoing degree of zero is a* sink

Lemma 3.1 *If G contains only one source or sink, its removal from G does not change $count(G)$.*

Having a single source in the DAG indicates that this vertex always occupies the first position in all topological orders. Hence, the source does not contribute to the cardinality of topological orders.

Lemma 3.2 *If there are m graphs $G_1, G_2, ..., G_m$ with $k_1 = |V_1|, k_2 = |V_2|, ..., k_m = |V_m|$ nodes, and $n = \sum_{i=1}^{m} k_i$, number of ways to linearly order them is the multinomial coefficient:*

$$\left(\begin{array}{c} n \\ k_1, k_2, ..., k_m \end{array} \right) = \frac{n!}{k_1! k_2! \ ... \ k_m!}$$

As explained in [7], this is the same as the number of different ways of splitting a set of n elements into m disjoint subsets, where each subset is an ordered set and the i^{th} subset has $k_i \geq 1$ elements.

To understand how this lemma is concluded, we study a situation where there are only two graphs. The reasoning can easily be extended to more. Assume there are two graphs with k_1 and k_2 nodes and both of them represent a chain, i.e. there is only one topological order for each. To calculate the number of ways to linearly order them, first we consider them as graphs without any edges. Therefore, this number would be the permutations of $k_1 + k_2$ nodes, i.e. $(k_1 + k_2)!$. Now, we have to rule out $k_1!$ orders,

because the number of possible permutations between k_1 nodes, based on the edges we ignored at first, is 1, hence we have: $\frac{(k_1+k_2)!}{k_1!}$. The same concept applies to eliminate $k_2!$ orders between k_2 nodes: $\frac{(k_1+k_2)!}{k_1!k_2!}$. For detailed proof, refer to [7].

Lemma 3.3 *If G is the disjoint union of m graphs $G_1, G_2, ..., G_m$ with $|V_1|, |V_2|, ..., |V_m|$ nodes respectively, since the G_i graphs are independent of each other, we have:*

$$\begin{aligned} count(G) &= \frac{(\sum_{i=1}^{m} |V_i|)!}{\prod_{i=1}^{m} (|V_i|!)} \prod_{i=1}^{m} count(G_i) \\ &= \frac{(|V_1| + |V_2| + ... + |V_m|)!}{|V_1|! \; . \; |V_2|! \; . \; ... \; . \; |V_m|!} \; . \; count(G_1) \; . \; count(G_2) \; . \; ... \; . \; count(G_m) \end{aligned}$$

This is an extension to the previous Lemma, where the number of topological orders for each graph is not necessarily 1. Which is why, the equation consists of their products.

Lemma 3.4 *Given G with at least one edge, say e_{ij}, two graphs may be derived:*

1. *G' is the graph obtained by removing e_{ij} from G*
2. *G'' is the graph obtained by replacing e_{ij} with e_{ji}, i.e. reversing the edge*

It then follows that:

$$count(G) = count(G') - count(G'')$$

If there is an edge from vertex a to b, it means in every topological order a occurs before b. By removing the edge, half of the times a occurs before b, and for the other half b occurs before a. Hence, "number of topological orders in which a can appear before or after b", minus "number of topological orders in which b must occur before a" is the same as "number of topological orders that a must occur before b". One thing to consider is that reversing an edge in G can sometimes produce a G'' containing a cycle. It logically follows that in such cases, $count(G'') = 0$.

3.4.2 Implementing the Counter

Using the above lemmas, we can implement several strategies in order to decompose G to simple graphs (such as single node graphs or graphs without any edges) to easily evaluate $count(G)$. We can combine these strategies into a recursive divide and conquer algorithm guided by the following items:

1. Reducing the number of sources by transforming the graph using Lemma 3.4 and Lemma 3.1.
2. Identifying disconnected components.
3. Delegating the evaluation of the components to a recursive step.
4. Combining the results of the delegation using Lemma. 3.3

We explain how the calculation of *count* is performed by means of an example. Let us assume a graph like G as depicted in Figure 3.5. The cardinality symbol surrounding a graph ($|G|$) indicates the number of topological orders for that graph. The $\otimes$ operator indicates that the number of topological orders from disconnected components are combined using Lemma 3.3.

There are 2 sources, a and b, in graph G_0. Our goal at this stage is to reduce the number of sources to 1. Then, by using Lemma 3.1, we can reduce the size of the problem. To achieve this, we use Lemma 3.4 on edge e_{ac}. The reason to select e_{ac} is the fact that if there are multiple sources, we intend to make the source with higher outgoing degree the single source[7] so that when it is removed (based on Lemma 3.1), the likelihood of having disconnected sub-graphs increases. The outgoing degree of source b is higher than the outgoing degree of source a. In next steps we will see that removing the single source b divides the graph into 3 disconnected sub-graphs. According to Lemma 3.4, we have to calculate $count(G_1)$ and subtract $count(G_2)$ to achieve $count(G)$.

Graph G_1 consists of two disjoint sub-graphs. One of these two is a graph with the single node a with number of topological orders 1. The other sub-graph, G_4, has a single source which can be removed (Lemma 3.1) and then divided into 3 independent sub-graphs as shown in steps 2 and 3. Graph G_2 is also converted to smaller graphs. In Figure 3.5, all of the steps are shown in order.

After breaking the graph into simple graphs (such as single node graphs or graphs without any edges) the final answer can be obtained from them. In Figure 3.6, it is shown how the number of topological orders for these simple graphs are used together to build the final answer (90). For example, using Lemma 3.3, the $count(G_5)$ is $count(G_6) \otimes count(G_7) \otimes count(G_8)$ which is 30.

After getting acquainted with the concepts, we introduce the *counting* in Algorithm 3.5. The first part of the procedure is responsible for removing the single source/sink, and also finding simple patterns like a cyclic graph or a graph with no edges. Next, if the graph is a connected graph, Lemma 3.4 is applied. Reversing an edge which is itself the result of reversing another edge may lead to a vicious cycle. This is visually illustrated in Figure 3.7. That is why in our algorithm, all transformed edges are flagged so that they are not transformed again. If the input graph consists of two or more disconnected sub-graphs, they are handled separately and the results are combined (Lemma 3.3).

3.4.2.1 Complexity Analysis

The complexity of this algorithm depends not only on number of vertices but also how they are connected to each other. Necessarily, the growth in number of edges does not increase the complexity. Assume a graph with maximum number of edges possible without having a cycle. For such graphs, after removing redundant edges, as explained in next chapter, what remains is a graph with a single chain of edges which its topological order number can easily be calculated.

For our algorithm, the worst case happens when the input is a complete bipartite

[7] To do this, at least one edge of other sources must be reversed based on Lemma 3.4. Because of this reversed edge, they will have an incoming degree greater than zero and will not be a source any more.

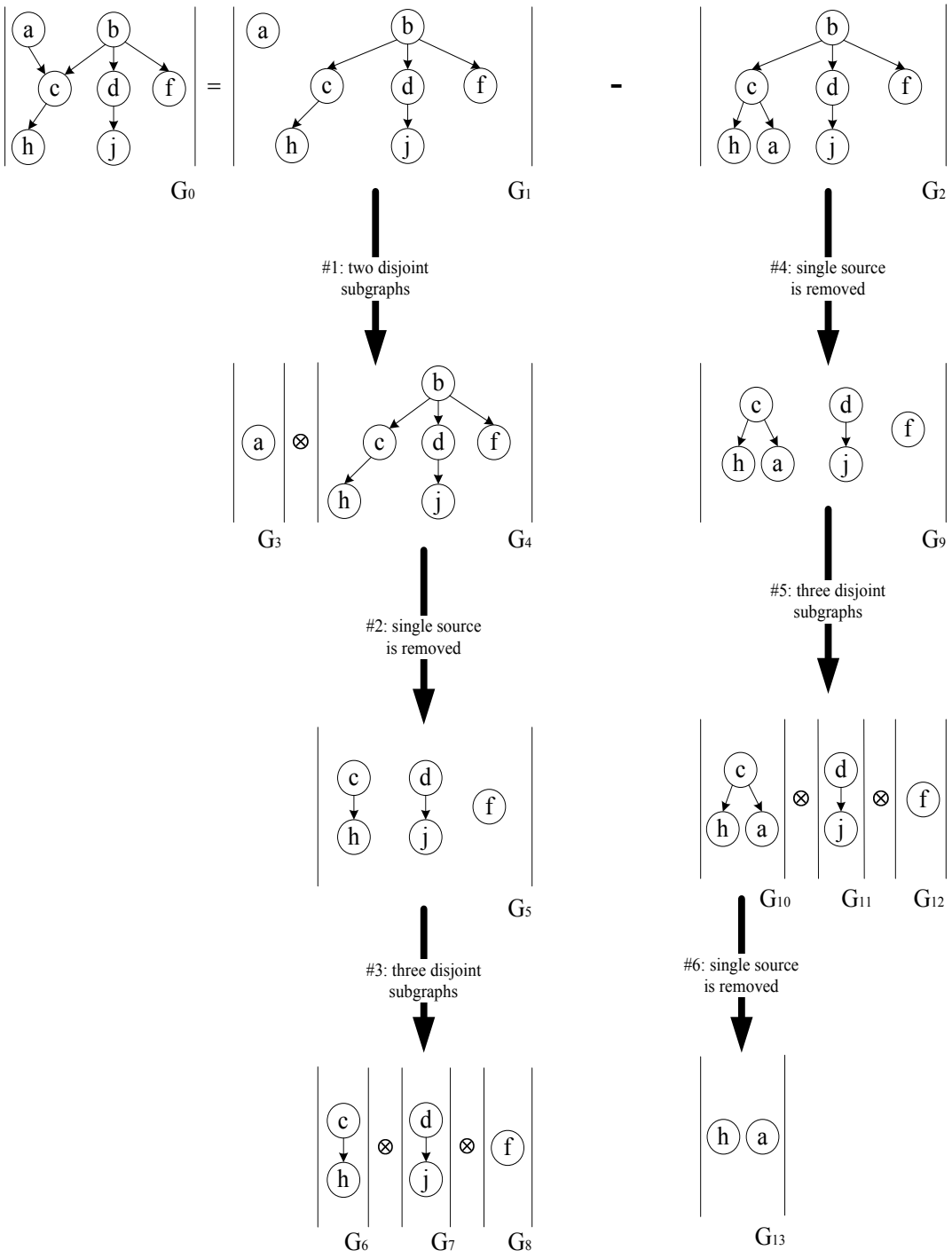


Figure 3.5: Transforming and dividing the problem into smaller manageable graphs in 6 steps (depth of 3 recursive calls).

Algorithm 3.5 Counting number of topological orders

```
function COUNT(G)
    while |sources| = 1 do                                          ▷ minimising
        G ← G − source-node
    end while
    while |sinks| = 1 do                                            ▷ minimising
        G ← G − sink-node
    end while
    if |V| ≤ 1                          ▷ If so, graph was a single chain of dependencies
        return 1
    end if
    if |sources| = 0                                                ▷ there is a cycle
        return 0
    end if
    if |E| = 0                                                      ▷ no edges in the graph
        return |V|!
    end if

    subgraphs ← ALL-DISJOINTED-SUBGRAPHS(G)
    if |subgraphs| = 1
        sort sources on ascending output degree
        for all s ∈ sources do
            for all e ∈ edges[s] do                  ▷ visits edges in no specific order
                if e is an already reversed edge
                    continue
                end if
                p ← count(G without edge e)                         ▷ recursion
                q ← count(G with reversed edge e)                   ▷ recursion
                if p = −1 or q = −1                  ▷ no answer on transformed graphs
                    continue
                else
                    return p − q
                end if
            end for                          ▷ looping over the edges of current source s
        end for                                              ▷ looping over the sources
        return −1                                ▷ nothing was returned as an answer
    else                                                    ▷ there are disconnected components
        m ← 0
        total-orders ← 1
        for all g ∈ subgraphs do
            n ← number-of-nodes[g]
            q ← COUNT(g)                                            ▷ recursion
            total-orders ← ((m+n)!)/(m!n!) × total-orders × q                ▷ combining
            m ← m + n                                ▷ growing till covers all components
            if total-orders = 0                      ▷ this may happen because of cycles
                break                                                ▷ do not waste time
            end if
        end for                                  ▷ looping over disconnected components
    end if
    return total-orders
end function
```

graph. A bipartite graph is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V. In complete bipartite graph for any two vertices, $u \in U$ and $v \in V$, there exists a (u, v) edge in the graph. A complete bipartite graph is notated as $K_{m,n}$ where m and n are the number of vertices in U and V respectively. A $K_{3,2}$ graph is depicted in Figure 3.8. In our worst case, U is the set of sources and V is the set of sinks. The number of vertices in these sets are either the same or their difference is one. The recursions in our algorithm grows exponentially for such cases, while the number of topological order for a $K_{m,n}$ graph can be calculated from $m!n!$.

To use the counting algorithm efficiently, we apply it on a subset of the input graph. This is explained in next section.

3.4.3 Using the Counting Algorithm on a Sub-graph of DDG

If the number of topological orders for a sub-graph of the DDG is bigger than the iteration number, we can select our random strategy (swapping and landing) over the brute-force. What we need is an algorithm that selects this sub-graph and apply the counting on this smaller graph.

Algorithm 3.6 proposes an approach which results in efficient use of the counting function. It starts with removing all source nodes from the DDG and adding them to an initially empty sub-graph. At this point, the `count(sub-graph)` is $s!$, where s is the number of sources. If the count is bigger than the `number-of-iterations`, the random strategy is selected and the algorithm finishes. However, if the count is less than `number-of-iterations`, we remove another node from DDG and add it to the sub-graph. This new node, must be a new source in DDG (after removing previous sources). Moreover, if there is any edge from *previously added nodes* to this *new node* in *initial DDG*, these edges are also generated while adding this node to the sub-graph. This guarantees that the `count(sub-graph)` never decreases while the size of sub-graph is increasing. Next, the `count(sub-graph)` and `number-of-iterations` are compared again. If the count is bigger, the random strategy is chosen and the algorithm finishes. If not, another node is removed and added to sub-graph. This goes on until either the count becomes bigger than `number-of-iterations` or there is no more node to add. In later case, the brute-force is selected as the strategy of the iterative scheduling.

3.5 Summary

The core approach used by the iterative scheduler is the *swapping* strategy which utilises a directed search. This approach enables modifying the instruction priorities and perform a local search for schedules. It starts with the priorities that are obtained from the scheduling heuristics and results in a fairly good schedule. As detailed in Chapter 5, it has been observed that this approach finds better solutions very fast.

To give the iterative scheduler the opportunity to find the global minimum, the random landing method was implemented. This method is an approach that completely randomises the schedules while respecting the DDG, so that every schedule has the same chance of being produced.

Algorithm 3.6 Deciding the iterative scheduling strategy

```
Require: G is the DDG of a basic block
  function STRATEGY(G)
    G' ← G
    sub-graph ← REMOVE-ALL-SOURCES(G')
    repeat
      if count(sub-graph) > number-of-iterations
        return SWAPPING-STRATEGY
      end if
      if G' = empty                          ▷ whole graph is covered
        return BRUTEFORCE-STRATEGY
      end if
      sub-graph ← REMOVE-ONE-SOURCE(G')      ▷ node along its edges are added
    until TRUE
  end function
```

Although counting the number of topological orders for every basic block added an overhead to the iterative scheduler, it reduced the compilation time in all. This was because of small basic blocks which had a few valid schedules. Without the *counting*, there used to be lack of accountability about how much of the solution space was covered, but with the *counting*, the rest of iterations were skipped as soon as all schedules were produced. This also guaranteed finding the best answer for those cases, since the whole solution space was explored.

Three topics were introduced in this chapter: swapping, landing and counting. Figure 3.9 depicts an overview of how the iterative scheduler makes use of them.

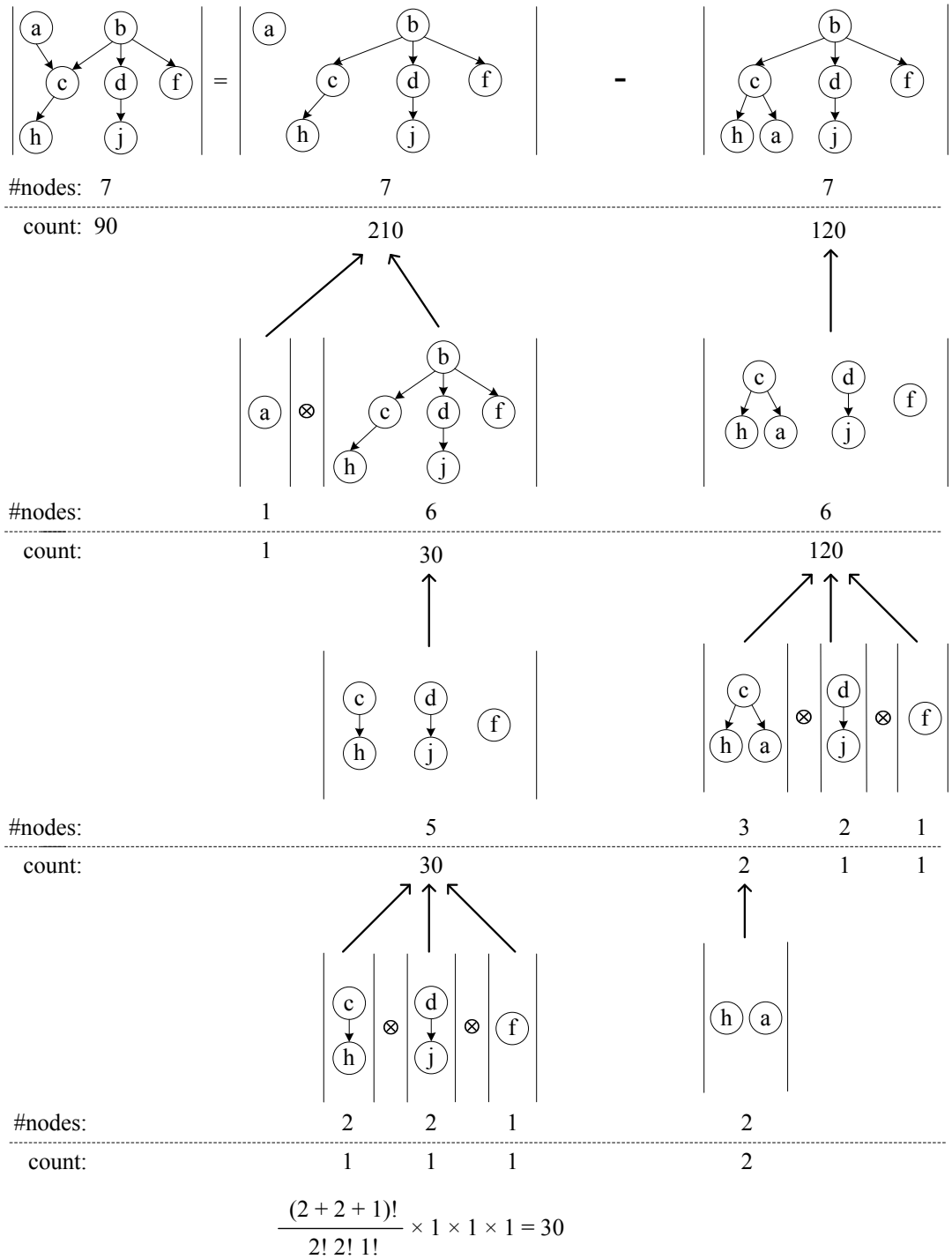


Figure 3.6: Finding the answer by solving smaller problems at the bottom and going to the top.

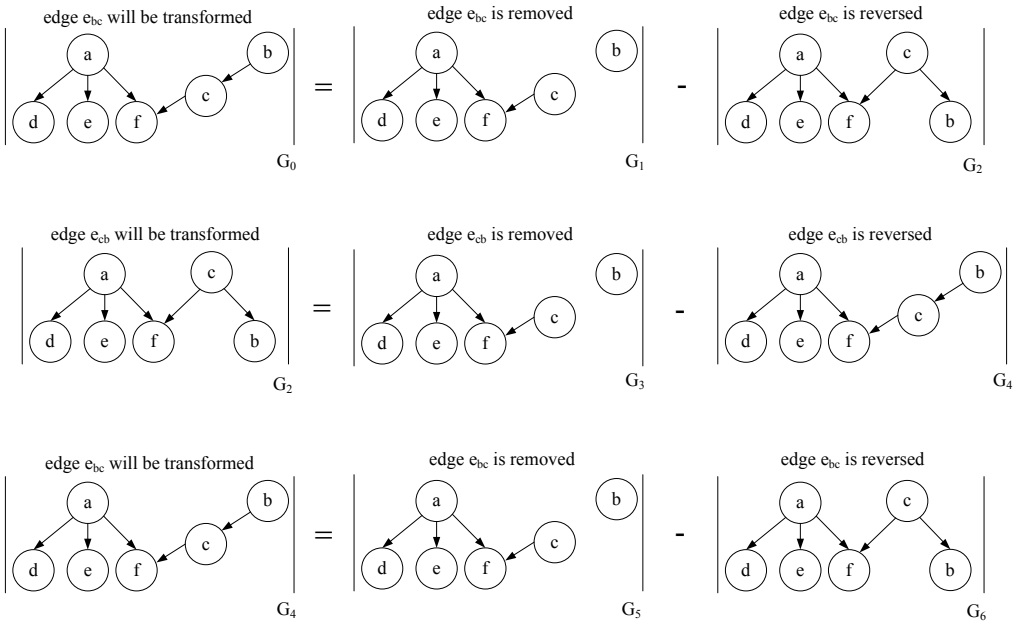


Figure 3.7: Racing of e_{bc} and its reversed version, e_{cb}, for transforming the problem, leads to a vicious cycle.

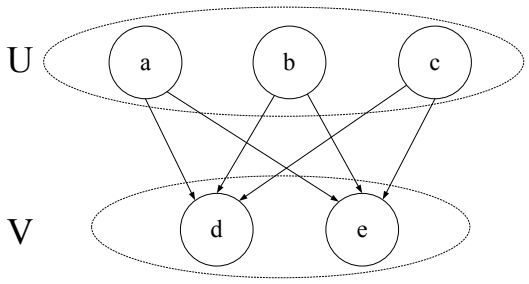


Figure 3.8: The $K_{3,2}$ bipartite graph.

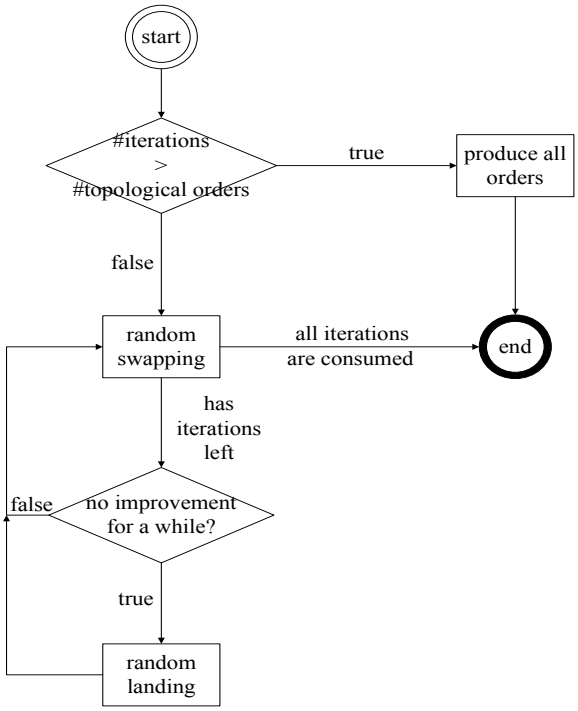


Figure 3.9: The strategy control flow for the iterative scheduler.

4 Iterative Framework

After studying the basics of instruction scheduling in Chapter 2 and introducing our iterative scheduling approach in Chapter 3, it is now time to present the framework which combines them. The iterative framework enables the compiler to schedule a basic block for a user-specified number of iterations which is defined in the compilation command-line. The framework also guarantees that the scope of each scheduling region is one basic block during the iterative scheduling. Moreover, it is possible to embed different strategies such as swapping or landing into the framework. This was accomplished by adding new interfaces to a few parts of the scheduling pass. Each interface, which is actually a function pointer, executes configurable functions at specific point of scheduling. Through these interfaces, a strategy can affect the list scheduler, evaluate the result, and decide what to do next. The strategy is encapsulated from the framework. This means that the framework does not need to know how the underlying strategy operates. Any strategy that relies on modifying the instruction priorities to affect the scheduler[1] can be used inside this framework.

In this chapter, first the scheduling pass in GCC[2] and its iterative version are discussed. Next, the information relating to each basic block that the framework provides such as the critical path, default performance (cycles of the non-iterative scheduled basic block), etc. are listed. Last but not least, a problem resulting in performance degradation is introduced and its solution is proposed.

4.1 High Level Overview of the Scheduling Pass

Before discussing about how to make the scheduling pass iterative, the non-iterative version must be reviewed. The current scheduling pass in GCC is implemented by Edelsohn et al. [4]. In Pseudo-code 4.1, the key steps of scheduling pass at the highest level are listed. The `init_region_infos()` function is actually a set of calls to functions initialising the basic blocks information which is used by the list scheduler. Afterwards, the compiler goes through every basic block and schedules their instructions. Finally, the wrapping of the scheduling pass takes place through a number of finalising functions which are grouped as `finalize_region_infos()` in the pseudo-code.

If we look into `schedule_region()` function, as listed in Pseudo-code 4.2, we see a few preliminary steps being executed before `shcedule_block()` begins scheduling the

[1] As illustrated in Section 2.3.4, by assigning different priorities to instructions, one may steer the scheduler. This means that the algorithm must be static, i.e. before the scheduling begins, it ought to have decided what to change.

[2] This pass is the same for both Sched1 and Sched2. The main differences between these two are: 1) Unlimited register file in Sched1; 2) Use of register pressure heuristic before any other heuristic in Sched1; 3) The scheduling scope in Sched1 is a region (consisting of at least one basic block) whereas in Sched2 it is only one basic block.

Pseudo-code 4.1: schedule_insns() in GCC.

```
/* scheduling for both Sched1 and Sched2 passes */
schedule_insns()
{
  init_region_infos();

  for (r=0; r < number_of_regions; r++)
    schedule_region(r);

  finalize_region_infos();
}
```

instructions. `compute_dependencies()` is one of these steps which builds the DDG from the instruction list. In `compute_priorities()`, critical path lengths are calculated and assigned to instructions as their priorities. Later, in `schedule_block()`, the instructions are scheduled in order of these priorities. Only after these steps, does the scheduling of instructions begin. Algorithm 2.1 in Chapter 2, details the `schedule_block()` function.

Pseudo-code 4.2: schedule_region() in GCC.

```
/* r is the region number */
schedule_region(r)
{
  /* using the region number to set up variables */
  ...
  compute_dependencies();     /* dependency analysis takes place   */
  compute_priorities();       /* calculating critical path lengths */

  /* in our case there is only one basic block */
  for (bb=0; bb < number_of_blocks; bb++)
    schedule_block(basic_block[bb]);
}
```

4.2 Iterative Scheduling

After briefly presenting the steps involved in the scheduling pass, we proceed by describing how this pass was transformed to an iterative scheduling pass. Later, the interfaces and information that are available in this iterative pass are explained.

The very first step in making the scheduling pass iterative is to invoke the initialisation and finalisation functions in between the iterations or otherwise the compiler would crash[3]. This is represented in Pseudo-code 4.3. The input of the scheduling pass is a list of instructions. The list scheduler works on this list and rearranges its elements.

[3]The reason behind this relies in the way that GCC performs its sanity checks which is out of our interest domain.

Hence, The output is the same list with its instructions rearranged. In order to invoke the scheduler repeatedly on the same instruction list, this input must be saved initially and restored in the beginning of each iteration. This, as listed in Pseudo-code 4.4, is the next step toward having a consistent iterative list scheduler.

Pseudo-code 4.3: Using the schedule_region() iteratively

```
schedule_insns()
{
  for (r=0; r < number_of_regions; r++)
  {
   /* region r is being scheduled for number_of_iteration times */
    for (i=0; i < number_of_iterations; i++)
    {
      init_region_infos();
      schedule_region(r);
      finalize_region_infos();
    }
  }
}
```

Pseudo-code 4.4: Adding save and restore to the previous iterative scheduler

```
schedule_insns()
{
  save_initial_insn_list();
  for (r=0; r < number_of_regions; r++)
  {
    /* region r is being scheduled for number_of_iteration times */
    for (i=0; i < number_of_iterations; i++)
    {
      restore_original_insn_list(); /* working on the same input */
      init_region_infos();
      schedule_region(r);
      finalize_region_infos();
    }
  }
}
```

4.2.1 Interfaces Provided by Iterative Framework

Being able to schedule a basic block repeatedly is not enough for us. Because the list scheduler is deterministic and the outcome will always be the same, unless some changes happen during each iteration. This was the reason that new interfaces were introduced to the iterative scheme. In order to perform certain tasks that are related to each strategy,

the iterative framework executes callbacks[4] provided by the strategy. Each callback, is executed at specific points during the iterative scheduling. Thus, the strategy can affect the list scheduler. The callbacks are:

- **pre-schedule:** This refers to the actions that must take place before each scheduling. Any strategy that wants to modify the parameters of the scheduler (especially the *priorities* of instructions) usually does it through this hook. This interface is listed as `iterative_pre_scheduling()` in Pseudo-code 4.6.
- **cost:** This executes the code responsible for cost evaluation of the currently scheduled sequence. The cost may be the number of clock cycles, but it can be anything such as code size, register pressure, etc. This is what the `iterative_evaluate_order()` in Pseudo-code 4.6 does.
- **post-schedule:** It is the function that is executed after each scheduling. Generally, one can decide the next step based on the implemented strategy at this point. In Pseudo-code 4.6, `iterative_post_scheduling()` represents this function.
- **break-predicate:** Through this callback, it is possible to evaluate a set of conditions that if satisfied, the framework will stop the iterative scheduling for the current basic block. For example, if the brute-force strategy specifies that all the orders are produced, there is no point in continuing any more. You see this callback with the name of `iterative_done_p()` in Pseudo-code 4.6.

Through these interfaces, it was made possible to affect the scheduler in each iteration, evaluate the results, and control the flow of iterative scheduling. During the `pre_schedule()` callback, the new priority of instructions are determined and the list scheduler uses them to construct another schedule. The `post_schedule()` hook keeps track of changes and their performance. For example, if there is a degradation in the new schedule, the swap strategy at this point (`post_schedule()`) signals the `pre_schedule()` callback to undo the last change and then make a new swap.

4.2.2 Additional Scheduling Information Provided by the Framework

The framework calculates and provides extra scheduling information to the embedded strategy. In this section, these information are listed:

4.2.2.1 Adjacency Matrix

The dependency matrices are the key data that the framework provides after analysing the current basic block. There are three different types of these matrices that make it easier to handle particular problems. The first one that will be explained is the *adjacency matrix* for the DDG. This matrix represents the dependencies between the instructions of a basic block, such that:

[4] A callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time. This is also called a *hook* in GCC.

$$dep\text{-}matrix_{ij} = \begin{cases} 1 & \text{if there is an edge in the DDG from } insn_i \text{ to } insn_j, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

4.2.2.2 Path Matrix

According to the DDG, this matrix indicates if there exists a path from one node to another. As illustrated in Equation 4.2, if there is a path from node i to node j, the corresponding element in the matrix is 1; otherwise it is 0. In other words, the 1's at row i represent the successors of $insn_i$ and the 1's at column j indicate the ancestors of $insn_j$. This matrix is used for the minimising the choices in swap strategy as described in Section 3.2.2. Moreover, this information is needed for building the next matrix (minimum precedence matrix) which reduces the number of edges in the graph so that the counting algorithm works faster on the DDG.

$$path\text{-}matrix_{ij} = \begin{cases} 1 & \text{if there is a path in the DDG from } insn_i \text{ to } insn_j, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

4.2.2.3 Minimum Precedence Matrix

As described in Section 3.4.1, a topological sort of a Directed Acyclic Graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right [2]. Not all of the edges in DDG are needed for the topological sorting. This concept is illustrated in Equation 4.3.

$$min\text{-}matrix_{ij} = \begin{cases} 1 & \text{if the only path from } i \text{ to } j \text{ is thorugh edge } e_{ij}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

To see how this equation can be useful, assume there is a graph with 3 nodes and $(a, b), (b, c)$, and (a, c) edges. (a, b) edge specifies that a must happen before b. Moreover, (b, c) specifies that b must occur before c. These two edges implicitly define that a must be before c. Hence, there is no need to keep the (a, c) edge.

The counting algorithm introduced in Section 3.4, handles graphs with less number of edges faster. Therefore, we wanted to have the minimum number of edges possible without loosing any useful information. Generating the *minimum precedence matrix* from the DDG, allows us to achieve this goal.

4.2.2.4 Other Extra Information

In addition to what we discussed, the framework provides more information to help the embedded strategy. The remaining extra data are:

- **Number of instructions:** The instructions in the basic block are counted and this value is stored in a global variable.
- **Critical path and its length:** After calculating the *priorities*, the maximum one is considered as the critical path length of the current basic block. This number represents the performance of an ideal schedule where there is enough resources available. Hence, it is a lower bound on the clock cycle for a basic block. This is why we also used it as a sanity check that the performance of each scheduling should not be less than this value.
- **Default cycle count:** For each basic block, before the iterative procedure begins, the basic block is scheduled with the default heuristics. The performance of this schedule is evaluated and set as the initial cost that must be improved.
- **Priority vector:** In the beginning, this array is initialised with the critical path length of instructions. During iterative scheduling, this vector represents the priority of the corresponding instructions (priority of $insn_i$ is the value of the i^{th} element). It is the sole factor that allows the strategy to alter the scheduling order. During the iterative phase, instead of calculating the *priority* of instructions, they are read from this array and assigned to the corresponding instruction. In the example shown in Section 2.3.4, we see that two different priority vectors, $\langle 3, 2, 2, 1, 1, 1, 1, 1\rangle$ and $\langle 2, 2, 3, 1, 1, 1, 1, 1\rangle$, resulted in different performances, 5 and 4 clock cycles respectively.

4.2.3 Other Framework Features

The framework does not only provide the strategies with useful information but also takes care of issues related to iterative scheduling. The key features are listed here:

- **Scheduling one block at a time:** The scope of each scheduling region is exactly one basic block during iterative scheduling. To have each region as one basic block, the `flag\_schedule\_interblock` in the compiler is set to 0. This is done only for iterative scheduling during *Sched1*, because already in *Sched2* each region is one basic block (refer to Section 2.2 for more details).
- **Last schedule:** After the last iteration, the order of instructions are determined by the last scheduling, which may or may not be the best order. Therefore, the current basic block is finally scheduled one more time with the parameters (*priorities*) which led to the best cost.
- **Handling solutions:** If a scheduling order better than the default is found for the current region (r) in *Sched2*, a flag (`load_priorities` in the region information is set as an indication. This leads to loading the same parameters which resulted

in this performance in *Sched3*. This is not true between *Sched1* and *Sched2*. If a better order is found in *Sched1*, only the output of this pass reflects this new solution and the parameters are not remembered.

- **Disabling the UID counter to save memory:** During the Register Transfer Language (RTL) passes in GCC, the information about each instruction is stored in a data structure called RTL Expression (RTX). These records contain information about the instruction they are representing such as source/destination operands, the cycle in which it was issued on, notes[5], etc. During basic block scheduling, some RTXs might get created (mostly they are instruction notes). Whenever an RTX is created, it is assigned a UID. This value is read from the `cur_insn_uid` variable. Afterwards, `cur_insn_uid` is incremented by one. GCC also considers this variable for handling its internal memory heaps. There is a linear correlation between the amount of memory allocated and the `cur_insn_uid`. The iterative framework momentarily stops any increase on this value. Moreover, to reuse some of the consumed space in the heap, the internal garbage collector of GCC is invoked after each iteration.

- **Brute-forcing when suitable:** If the number of iterations are big enough to cover all the possible topological orders of the DDG, the current strategy is replaced with a brute-force strategy. This is determined by the `set_strategy()` in `init_iterative_framework(r)` function, as listed in Pseudo-code 4.7.

4.3 The Framework's Final Design

Now that it is clear what the purpose of the iterative framework is, where it is used, and what it provides; it is time to see its implementation structure. In this Section, the code is studied in a top-down approach. We start with the highest level, and get into more details by viewing the code of the key functions.

As it is illustrated in Listing 4.5, if the compiler is asked to apply iterative scheduling for `number_of_iterations` > 0 times, the iterative framework takes over. The first step is to set the callback hooks. Here, the strategy is the swap strategy[6] explained in Chapter 3. The `swap_break` callback checks the conditions that if are met, the iterative loop terminates early. One of the conditions is the achievement of a cycle count equal to the critical path length. The other one is having no possible swaps. For instance, this can happen when the DDG is a single chain of dependencies. The `swap_pre_schedule` function, picks two instructions randomly for the priority exchange and the `swap_post_schedule` based on the performance of the schedule (evaluated by `clock_cycle_cost` callback), defines the next step. Please refer to Chapter 3 for more information about this strategy. After setting the hooks, the framework iteratively schedule the regions using the current strategy. If it is desired, one can set more callback hooks and try different strategies iteratively. Next, the `iterative_schedule()` function will be explained.

[5] "notes" are special annotations in RTX format that are attached to instructions as extra information.

Pseudo-code 4.5: schedule_insns() using iterative scheduling with swap strategy.

```
/* scheduling for both Sched1 and Sched2 passes */
schedule_insns()
{
  ...
  if (number_of_iterations == 0)
  {
      for (r=0; r < number_of_regions; r++)
        schedule_region(r);
  }
  else
  {
    /* setting the callback hooks */
    iterative_done_p           = swap_break;
    iterative_pre_scheduling   = swap_pre_schedule;
    iterative_post_scheduling  = swap_post_schedule;
    iterative_evaluate_order   = clock_cycle_cost;

    for (r=0; r < number_of_regions; r++)
      iterative_schedule(r);
  }
  ...
}
```

The iterative scheduler is invoked for each region (which is one basic block) separately. In the beginning, it takes extra actions and initialises variables. These are illustrated in Listing 4.7, such as saving the initial input, counting the number of instructions, enumerating them, building all the dependency matrices, finding the critical path and its length, and replacing the current strategy with the brute-force algorithm in case the number of iterations is big enough. The `set_strategy()` which is responsible for this decision, is the implementation of Algorithm 3.6. Further on, the list scheduler is invoked once to get the performance of the default schedule. Setting the global variable `iterative_phase` to true indicates that the next invocations of the list scheduler are part of the iterative framework in current scheduling pass. As a result, for example, the `compute_priorities()` function loads the priorities from the priority vector instead of calculating them.

After the initialising steps, the iterative loop starts which is repeated for `number_of_iterations` times. In the beginning of each iteration, the function hook `iterative_done_p()` checks the conditions defined by the current strategy and determines whether the loop must continue or not. If it is decided not to break the loop, the `INITIALIZE_REGIONS()` macro is used to make a function calls to set up regions information. This makes it possible to invoke the scheduler again. Right before calling the scheduling routine for the current region, the `iterative_pre_scheduling()`

They are generated and used by different optimisations passes.

[6]Mixed with landing approach to avoid local minima.

Pseudo-code 4.6: iterative_schedule(), the framework.

```
/* r is the number of region */
iterative_schedule(r)
{
  init_iterative_framework(r);  /* gathering needed information */

  /* Repeating till an early termination condition *
   * is met or we run our course                   */
  while (iter_counter < number_of_iterations)
  {
    if (iterative_done_p())    /* hook */
      break;
    iter_counter++;
    INITIALIZE_REGIONS();

    /* hook: Decides how the next order should look like */
    iterative_pre_scheduling();

    /* Running the scheduler on the region r */
    schedule_region(r);

    /* hook: Now, let's examine the result */
    iterative_evaluate_order();

    /* hook: Some house keeping for next iteration */
    iterative_post_scheduling();

    /* calls ggc_collect(), and restores saved initial input */
    FINALIZE_SCHEDULER();
  }

  /* calls the scheduler with the best parameters, etc. */
  finish_iterative_framework(r);
}
```

hook is invoked which allows the strategy to alter the parameters for scheduling. In `scheduling_region()`, the `compute_priorities()` loads the *priorities* from an array (which probably is manipulated by the pre-scheduling hook). Once scheduling is done, the performance of the schedule is evaluated by the `iterative_evaluate_order()` hook. Besides, this hook make it possible to save the current priorities if they led to shorter clock cycle. Next comes the last hook in this iteration, the `iterative_post_scheduling()`. According to the performance obtained, some steps might be taken such as setting the variable that makes the condition in `iterative_done_p()` to be true, or directing the next `iterative_pre_scheduling()`. At the end of the iteration, `FINALIZE_SCHEDULER()` macro invokes a set of functions. This allows using the `INITIALIZE_REGIONS()` again, calls the GCC's garbage collector, and restores the order of instructions. When the iterations are done, at `finish_iterative_framework()`, the

current region is scheduled once again with the best parameters observed.

Pseudo-code 4.7: Initialising the framework.

```
init_iterative_framework(r)
{
  save_initial_input();  /* remembering the rtx order */
  number_of_insns = count_instructions();
  if (number_of_insns == 0)
    return;
  enumerate_insns();

  /* getting three types of graphs out of the block */
  init_dep_matrix_count(&adj_matrix, &min_matrix, &path_matrix);

  /* finding the critical path and its length*/
  critical_path = get_critical_path();

  /* should we go on with current strategy or brute-force */
  set_strategy();

  /* scheduling with default heuristics */
  schedule_region(r);
  minimum = iterative_order_cost();

  FINALIZE_SCHEDULER();

  /* now we are actually in iterative mode */
  iterative_phase = true;

  return;
}
```

4.3.1 Promoting Instructions Leading to a Branch

In some cases, it has been observed that a found solution in *sched2* (a scheduling order other than the default one which has a better performance) can end up in more cycle counts in the end. In this section, an example of such cases is shown and the solution that tries to minimise this effect is discussed.

Assume a machine capable of issuing at most one multiplication, two addition, and one branch operations in each cycle. The number of clock cycles needed to finish multiplication and addition is one. The branch operation has a delay slot of three. Moreover, assume a basic block with a DDG as depicted in Figure 4.1. Each m_i is a multiplication instruction and each a_i is an addition instruction. The subscript i represents the UID. For each node, the number after the colon is the critical path length assigned to that instruction. Nodes with bigger values have higher importance over those with lower values. In case the *priorities* of two instructions are the same, the one with lower UID is

scheduled first. This DDG, considering the assigned *priorities*, ends up in a scheduling order with 5 clock cycles.

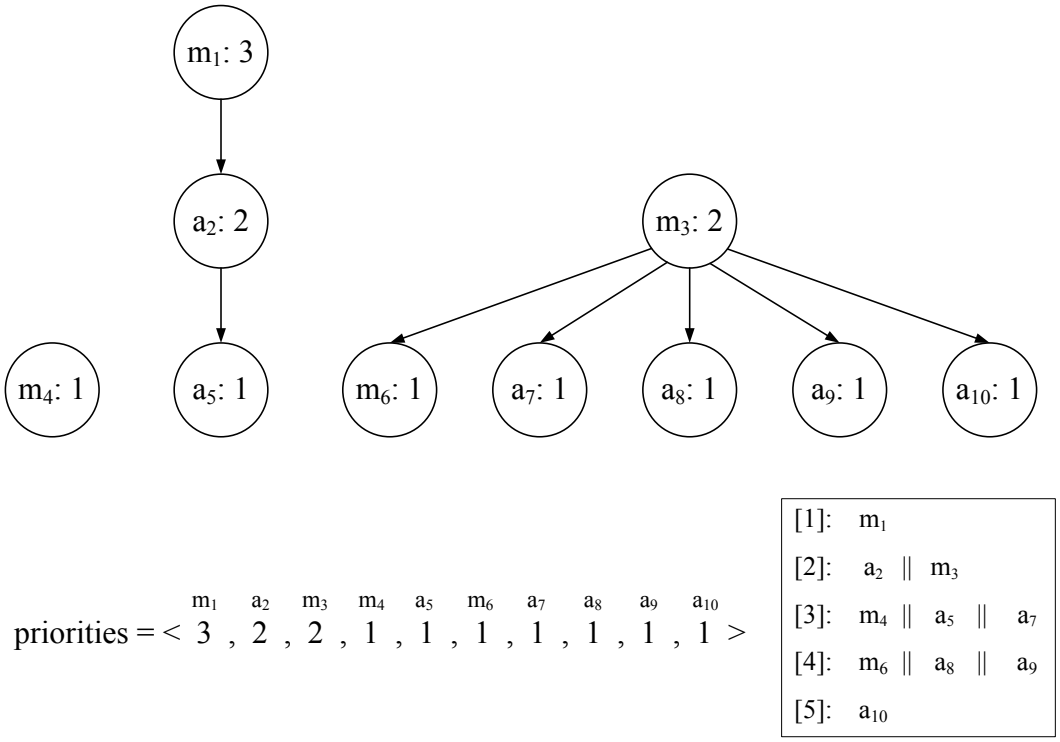


Figure 4.1: A DDG, its priority vector, and the schedule obtained using critical path heuristic.

There exists a better scheduling order than the one already seen. If the priority vector looks like the one in Figure 4.2, the performance is going to be 4 clock cycles. It is easy to see that this new priority vector is actually the original one with priorities of m_1 and a_2 swapped with the priorities of m_3 and m_6 respectively. At this point, the second schedule is better than the first.

During *Sched3*, the hardware loop instructions are introduced while its delay slots are being filled. These are not visible in *Sched2*. Assume in *Sched3*, a loop instruction is generated which depends on m_4. For example, the multiplication result of m_4 is written to register $r1$ and the"`loop bb5, r1`" instruction (wants to execute the fifth basic block, $r1$ times) needs to read that. After issuing the loop instruction in these two different scheduling orders, the performance turns the other way around. The order with 5 clock cycles will become 7 cycles and the order with 4 clock cycles will become 8 cycles, as depicted in Figure 4.3. This is because in the second order, which is obtained randomly, m_4 is issued one cycle later.

To avoid this situation as much as possible, a slight change has been made. In *sched3*, after loading the saved *priorities* for a basic block, the priority of instructions leading to a branch is increased. It is like loading almost the same order from *sched2*, while

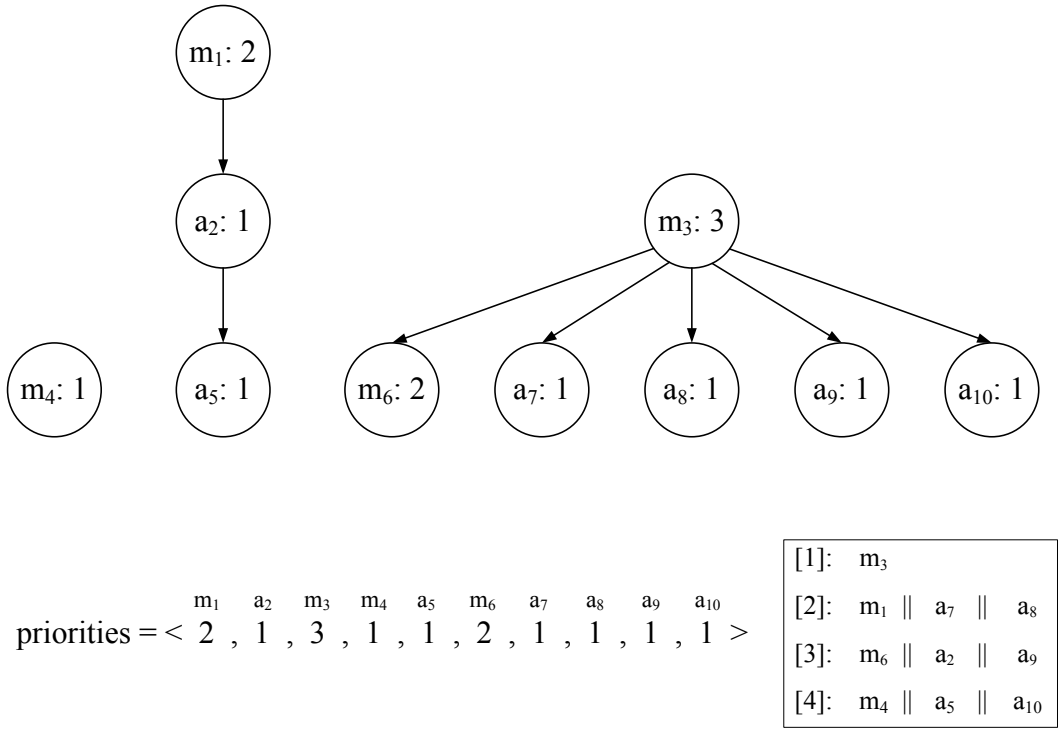


Figure 4.2: A DDG, its priority vector, and the schedule obtained using random priorities.

shifting some instructions above in the final scheduling sequence. Figure 4.4 illustrates an example of this. It must be mentioned that this solution rarely ends up in a worse scheduling order. For instance, because of shifting up some instructions, the critical path instructions might be postponed (due to the shortage of functional units). This approach, at first, was only used for the basic blocks that the iterative scheduling improved during *sched2*[7]. Since the result was satisfactory, it was decided to make this method the default behaviour of the compiler, disregarding the fact whether or not the iterative scheduling is happening. In other words, after determining the priority of instructions in *sched3*, the priority of predecessors of a branch instruction are increased. The determination of priorities can be done either by calculation or by loading them from an array.

4.4 Iterative Compilation Flags

Now that the iterative scheduling concepts are discussed, the compilation flags can be explained:

- **-msched1-iterative=** *number*: This flag sks the compiler to iteratively schedule

[7]So that in *sched3*, instead of letting the scheduler calculate the priorities from the DDG, it would read it from the saved priority vector.

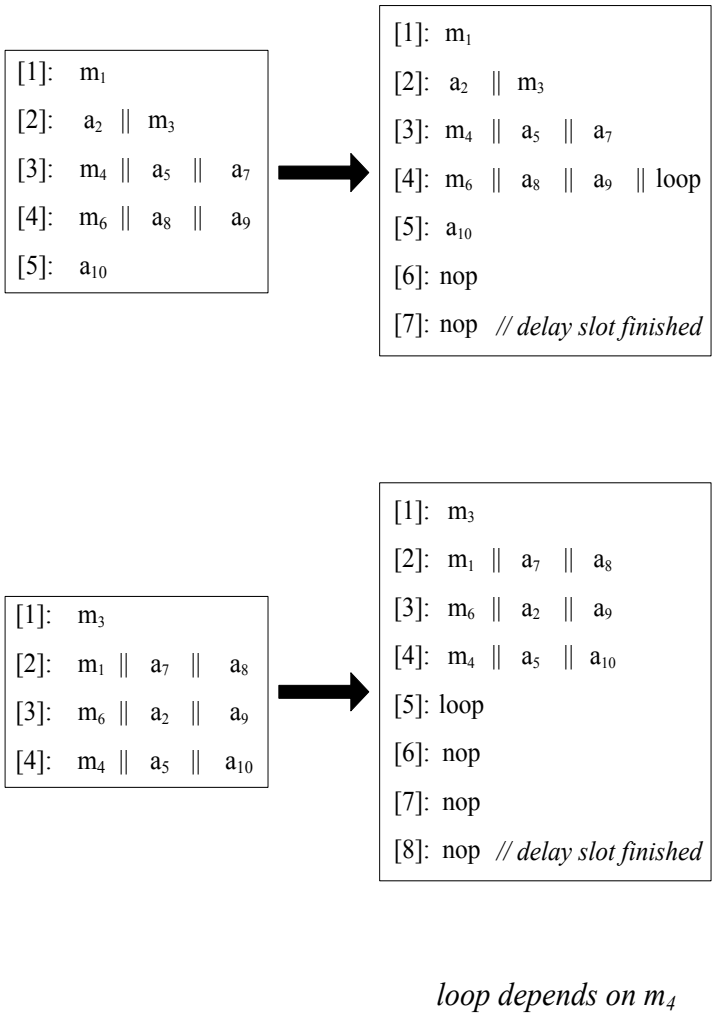


Figure 4.3: Two scheduling orders of the same basic block, before and after delay slot filling.

each basic block in *sched1*, for *number* times ($number > 0$), and selects the best.

- **-msched2-iterative=** *number*: The compiler will schedule each basic block during *sched2*, *number* ($number > 0$) of times. The one with the better performance will be the output of *sched2* and its order is going to be reproduced in *sched3*.

- **-fno-promote-branch-deps:** Since promoting the rank of instructions which lead to a branch in *sched3* has become the default action in the compiler, this flag will disable it. Nevertheless, when a non-default scheduling sequence is being transferred from *sched2* to *sched3*, the promotion takes place anyway.

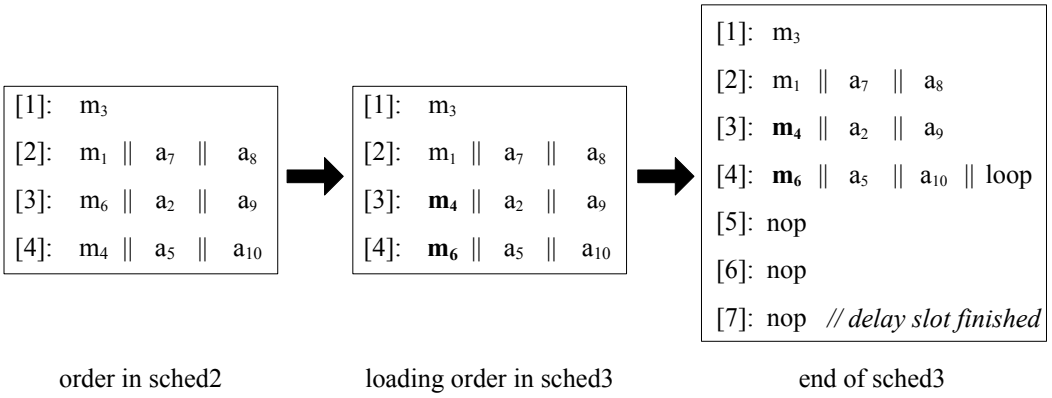


Figure 4.4: Loading the order from sched2 while giving higher importance to m_4.

- **-fsched-random-seed=** *number*: If the underlying strategy in iterative scheduling is using PRNGs, the random seed is initialised with *number*. In case this value is not given, the compiler uses a hard-coded number as the random seed. This guarantees a deterministic output from the compiler.

4.5 Summary

All the three scheduling passes in GCC use the `schedule_region()` function to schedule each region. It is not possible to call this function repeatedly on the same region, unless some extra initialising and finalising procedures are invoked in between. The iterative framework not only takes care of this function but also provides each iteration with the initial input. Moreover, it gathers useful information from the current basic block and makes them available. This pluggable iterative framework is designed in such a way that it remains remain encapsulated from the strategy.

The iterative scheduling may happen in *sched1* and/or *sched2*. It is not implemented for *sched3*, because *sched3* is already an iterative pass and it is not a good idea to apply another iterative approach on top of it. In *sched2*, when a better scheduling sequence is found, it becomes the output of the scheduling pass. Moreover, the same order is going to be reproduced in *sched3*. On the other side, if a better scheduling order is obtained in *sched1*, it is only going to be the output of that pass because it is desired have the effect of this sequence on the register allocator only, not to reproduce it again later.

There are scenarios where a winning solution (the scheduling sequence with a better performance than the default) in *sched2* ends up being a losing solution in *sched3*. This comes from the fact that different orders do not fill the delay slots in the same way and the information about these delay slots are not known at *sched2*. To minimise the effect of this lack of information, the instructions leading to a branch get higher priorities in case of loading an order from *sched2* in *sched3*.

Although the strategy embedded in the framework might use a pseudo-random approach, the output will still be the same as long as the input file and the compilation

flags are the same. This deterministic property allows the iterative scheduler to have a reproducible output.

5 Experimental Results

The DSP-IC group of ST-Ericsson, developed in-house test cases for checking the validity and the performance of the compiler. Most of these tests are DSP programs responsible for handling 2G (GSM), 3G (WCDMA), 4G (LTE), and wireless communications. Each test case listed in this section represents one of the functions in these test cases. The corresponding performance is obtained from counting the clock cycles at execution time. Two sets of benchmarks are considered for performance evaluation in this project[1]:

- **kernels-321:** This benchmark consists of 321 test cases in all. To measure the iterative scheduling improvement, the Iterative List Scheduler (ILS) was compared against the List Scheduler (LS) with `-O3` and `-fno-ivopts`[2] as common compilation flags ("`-O3 + -fno-ivopts`" vs. "`-O3 + -fno-ivopts + iterative flags`").

- **kernels-81:** This is a set of test cases that are representative of EVP's workload. It consists of 81 test cases. A combination of flags, called *sweep flags*, is selected for each test case to tune it in such a way that the best cycle count is achieved. In this benchmark, the improvement of ILS is measured by adding the iterative flags to the sweep flags (`sweep flags`" as reference vs. "`sweep flags + iterative flags`").

5.1 "kernels-321" Benchmark

In this benchmark, 321 test cases were compiled with GCC. The "`-O3 + -fno-ivopts`" flags were used for the reference (LS). Then once again, the test cases were compiled with the iterative flags in addition to the reference flags[3] (ILS).

In Table 5.1, an overview of this benchmark is given. 53% of the tests (171 out of 321) were improved and the average (geometric mean) of total improvement was **3.67%**. On the side, the average code size was reduced by 1.20%. Originally, using the list scheduler, the compilation time was 4 minutes. This time increased to 34 minutes while using the iterative list scheduler. The summary of improvements is depicted in Figure 5.1 and Figure 5.2. For more details on the performance of each test case please consult the tables in Appendix A.

[1] They are not exclusive.

[2] Suppresses high-level loop induction variable optimisations, which are enabled if -O1 is used. These optimisations are generally profitable but, for some specific cases of loops with numerous uses of the iteration variable that follow a common pattern, they may end up destroying the regularity that could be exploited at a lower level and thus producing inferior code.

[3] For example the iterative flags for the "libEvpFft256" test case are: -fsched-random-seed=38265161 + -msched1-iterative=515 + -msched2-iterative=1600.

Cycle counts improvement	**3.67%**
Improved test cases	53%
Increased compilation time	850%
Code size reduction	1.20%

Table 5.1: Statistics of applying iterative scheduling on "kernels-321".

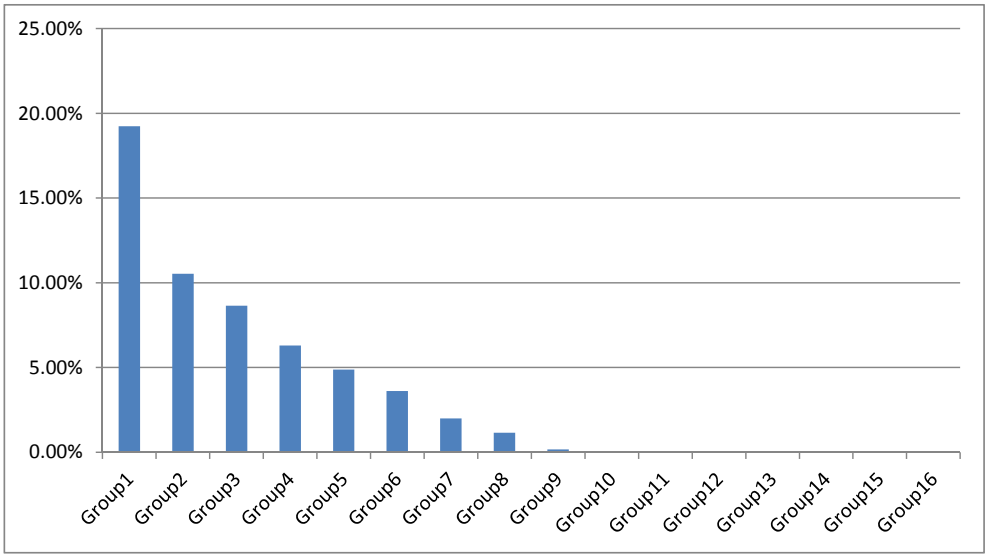


Figure 5.1: Geometric mean of improvements for each "kernels-321" benchmark group.

5.2 "kernels-81" Benchmark

In this benchmark, 81 test cases were compiled with GCC. The iterative scheduler ("`sweep flags + iterative flags`") was compared against the reference non-iterative scheduler (only "`sweep flags`"). The goal in this benchmark was to improve the best results obtained by the non-iterative compiler. The iterative scheduler managed to improve 40% of the benchmarks (33 out of 81). The geometric mean of the total improvement was **1.03%**. This was accomplished by using the iterative scheduling in *sched1* and/or *sched2* with an average of 5000 iterations. The performance comparison between the List Scheduler (LS) and the Iterative List Scheduler (ILS) is listed in Table B.1 and Table B.2.

In Figure 5.3, the histogram of average improvements is depicted. The first group consisting of "conven", "cowc", "libEvpFft256", etc. has an average improvement of 5.966%. Figure 5.4 categorises the improvements into different ranges and demonstrates how many of them fall under the same range.

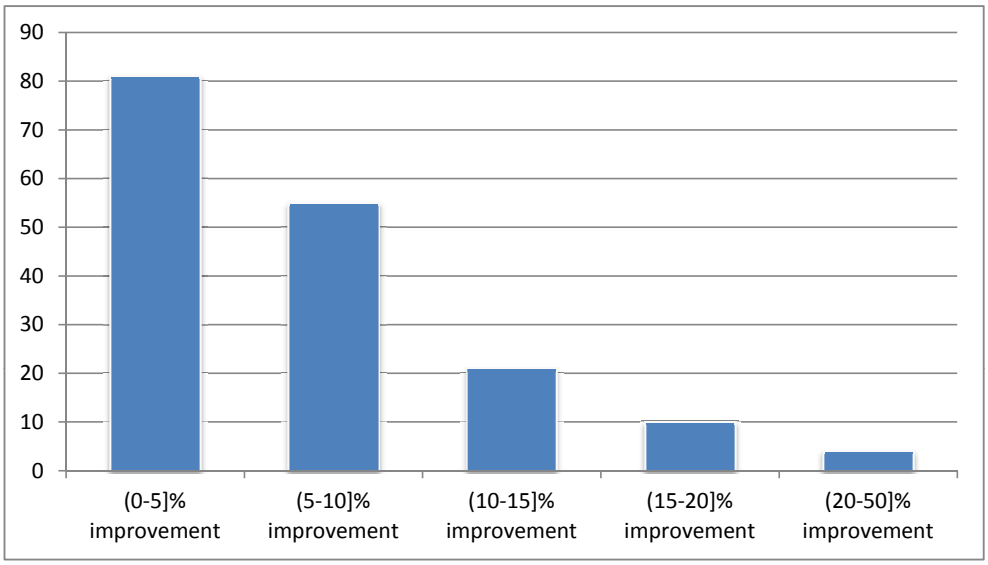


Figure 5.2: Distribution of 171 improved test cases in "kernels-321" over different improvement ranges.

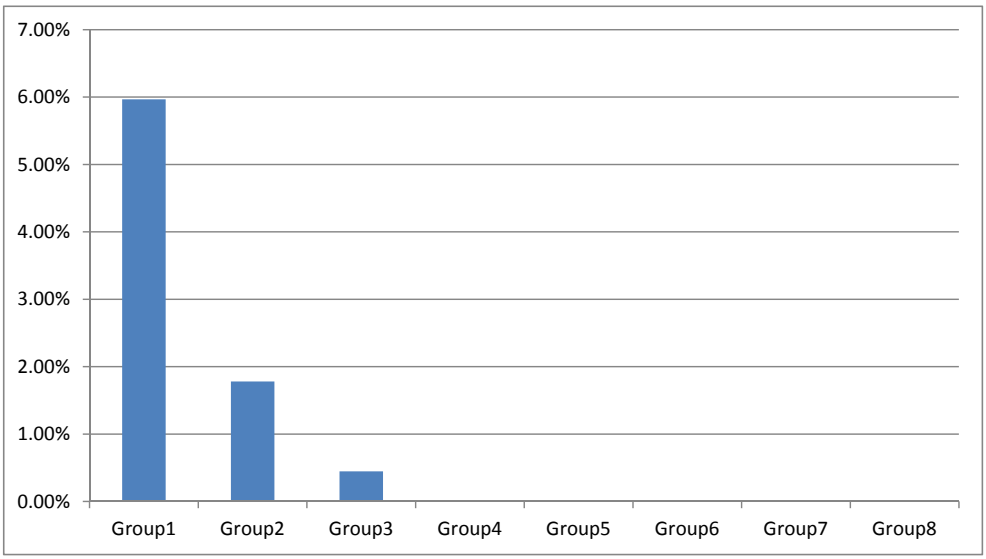


Figure 5.3: Geometric mean of improvements for each "kernels-81" benchmark group.

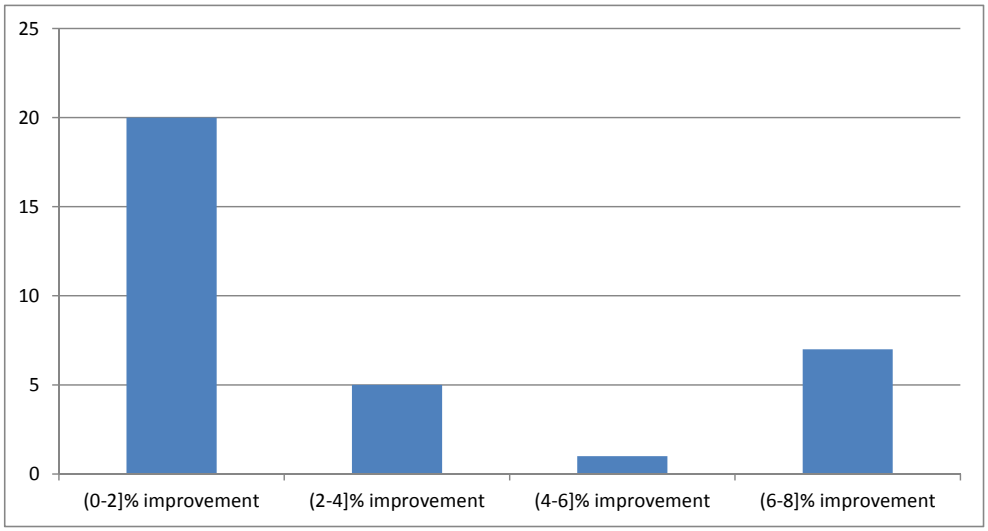


Figure 5.4: Distribution of 33 improved test cases from "kernels-81" over different improvement ranges.

5.3 Compilation Time

The 3.67% performance improvement for the "kernels-321" was achieved after 34 minutes of compilations, whereas the non-iterative version took 4 minutes. In Table 5.2, ten test cases of the benchmark are listed along with corresponding compilation time for both LS and ILS. For instance, it can be seen that the `golay_manoverlay` test case achieved 2 times speed-up with an extra minute compilation time.

Test Case	LS Compilation Time	ILS Compilation Time	Number of Iterations	Improved Cycles
cowc_fxp_kernel	0.93s	2.21s	174	33.59%
cowc_flp_kernel	0.86s	13.57s	1061	10.89%
golay_manoverlay	1.48s	61.42s	2495	49.84%
libEvpMatInvSqrt	1.16s	23.16s	1048	22.22%
libEvpTranspose	0.61s	5.06s	408	15.36%
libEvpFft256	2.08s	107.32s	2115	9.65%
libEvpFft256_div2	1.27s	61.28s	1861	8.54%
libEvpFft256_sat	2.15s	102.70s	2115	9.65%
mc_16xN_b	0.60s	11.82s	2194	7.32%
demuxb_2	0.62s	4.99s	1939	17.07%

Table 5.2: Compilation times for a selected set of test cases from "kernels-321": LS vs. ILS.

5.4 The COmbining Weight Computation (COWC) benchmark

Since the 2G standard is well established after years of implementation, the performance of various multi standards modems in the market over this standard is more or less the same. This is also true for the 3G standard. The key performance differentiator for these modems comes from the implementation of the new 4G (LTE) standard. The EVP, as a part of the Thorium modem, executes the software partition corresponding to the LTE standard.

The biggest component of the LTE load on the EVP is the adaptive filter, COWC. The COWC kernel discussed in this section, is the of the COWC listed in the "kernels-321" becnhmark. It accounts for 33% of the EVP's workload and is invoked 13K times a second. Thus, according to Amdahl's law, it is a primary objective when attempting to improve performance. We have found that the compiler achieved producing a **10%** shorter version of the COWC kernel after applying the iterative scheduling.

Thorium drains a current of 118 mA of which the EVP accounts for 25 mA. After improving the COWC with iterative scheduling, 1 mA was reduced, resulting in 4% reduction of the current drain for the EVP and consequently almost 1% for the entire modem. This happened because the reduced cycle count of the running application allowed the processor to finish its tasks earlier and switch to stand-by mode.

In Table 5.3, the performance of COWC is listed after compilation with different flags. Moreover, the utilisation of the VMAC unit as the mostly used resource in this adaptive filter is shown. The *sweep flags* in the table refers to a set of compiler optimisation flags used for the test case such that the minimum cycle count is achieved.

Compilation Flags	Clock Cycles	VMAC Utilisation
`-O3`	91	45.65%
`-O3` + sweep flags	81	51.85%
`-O3` + sweep flags + iterative flags	73	57.53%

Table 5.3: Static clock cycle counts of COWC kernel with different flags.

6 Conclusions and Future Work

This research was undertaken in DSP-IC group of ST-Ericsson and its goal has been to reduce the cycle count of compiled programs. These programs are mostly responsible for handling 2G, 3G, 4G standards and are compiled by a GCC port for an embedded VLIW processor, namely the EVP. To adhere to simplicity and power efficiency, EVP relies on the compiler to increase its throughput.

To achieve our goal, the scheduling pass in GCC, out of many optimisation passes, was targeted. We introduced a random iterative scheduling algorithm to this pass. In this way, different schedules are produced and the best one is selected.

The iterative scheduler proposed in this thesis was added to the production compiler and is currently being used as a part of the speed-up optimisation settings. The value and portability of the iterative scheduler was empirically proven by several typical test cases, especially when it managed to improve the updated COWC filter by 10%.

6.1 Conclusions

In this section, the conclusions that were drawn after this research are briefly listed.

- The scheduling heuristics in GCC are used for a long time, while being tweaked to achieve fairly good, if not optimal, schedules. To benefit from this, we use this schedule as the initial point upon which a local search (swapping) is performed to find better solutions.

- Our iterative scheduler modifies the priority of instructions, in order to produce different schedules. These priorities are mapped into a vector, called priority vector, such that changes to this vector affect the output of the list scheduler. It is important to initialise the priority vector with numbers that are likely to produce good schedules. We had two options in that matter. The first one involved using *unique* numbers that represent the position of the corresponding instruction in the default schedule. For example, the priority value for the last instruction in the default schedule would be 1, the priority of the second to last instruction would be 2, and so on. The second option was initialising the priority vector with the Critical Path Length (CPL) of instructions which are not necessarily unique. We selected the second option, because when priorities are not unique, other scheduling heuristics have a chance to be applied and construct a better schedule.

- The iterative scheduler repeatedly modifies the priority vector. These modifications are done on top of each other. To speed-up the process of finding better schedules, we tweaked our iterative scheduler to take into account the outcome of changes to the priority vector. Whenever a change to the priority vector results to more cycle

count for a scheduled basic block, that change is discarded and another change is made (going toward some other solution).

- Our iterative scheduler, performs a local search for other schedules. Two schedules are considered local to each other when their instruction orders are similar. Having a schedule (and its priority vector), we want to take a small step in the solution space (all valid schedules) to find another schedule. If the changes on the priority vector are too much, the corresponding schedule can be very different from the current one. What our *random swapping* approach does in each iteration is selecting two instructions randomly and exchanging their priorities. This way, the changes are under control and we can gradually step toward better schedules, especially when it is combined with a directed search as described above.

- A strategy for getting out of local minimum traps had to be developed because of using a directed search method. This strategy, had to produce any valid schedule without being biased. The landing algorithm described in Section 3.3 is as such.

- There is no need to perform a random search for basic blocks with small number of valid schedules. Instead, all of the schedules must be produced. This has two advantages. First, since all of the schedules are produced, the best solution is found. Second, this can decrease the compilation time. For example, if the iterative scheduler is asked to perform 1000 iterations while the number of possible schedules for a DDG is 20, the iterative scheduler produces the 20 schedules and skips the rest of iterations. To do this, the iterative scheduler needs to calculate the number of valid schedules or at least should know if this number is bigger than the given iteration number. This is what the counting algorithm in Section 3.4 helps us to do.

- Adding the iterative scheduler to GCC resulted in speed improvement for the compiled applications. The applications that were benchmarked are mostly responsible for handling mobile signals and wireless communications. In one of the benchmarks, our iterative scheduler moderately improved the `-O3` optimisation. When we combined the iterative scheduler with the `-O3` optimisation, a 3.67% cycle count speed-up was achieved, compared to the benchmark where only `-O3` flag was used. The compilation for the iterative scheduler took 34 minutes, while the non-iterative version was only 4 minutes. While this might not be acceptable for day to day compilations, but it is well received in ST-Ericsson, since they want to improve their critical and limited programs. In another benchmark, we took the fastest version of applications[1] and observed that the iterative scheduler still managed to improve their cycle count by 1.02%.

[1] A set of flags for each application was used, called the sweep flags, to produce binaries with smallest cycle count observed.

6.2 Future Work

In this section, we present some of the issues that were not considered in this work. They can be interesting aspects to take into account when taking this work further.

- Instead of applying and evaluating the iterative scheduling in the same pass, one can widen the scope of the iteration. One good example would be changing the schedule in *Sched1* and evaluating the result in *Sched3*.
- A different metric than the clock cycles can be used for evaluating the performance of schedules in *Sched1*. For instance, this new metric can be the average of register pressure per basic block. It is also possible to focus on other aspects of improvements such as code size.
- The performance of the *counting* algorithm discussed in Section 3.4 can be optimised. For example, one of the key points to be considered is the selection of the edge for dividing the graph. A well-devised criterion for selecting such edges will reduce the number of steps to solve the problem.

The iterative scheduling proposed in this research indicates once again the importance of performing randomised search for improving the solutions to NP-complete problems. The same concept can be used in various parts of compilation process such as clustering or software pipelining in which the optimal solutions are not known.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Prentice Hall, 2006.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 2001.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems (1991), 451–490.

[4] D. Edelsohn, W. Gellerich, M. Hagog, D. Naishlos, M. Namolaru, E. Pasch, H. Penner, U. Weigand, and A. Zaks, *Contributions to the GNU Compiler Collection*, IBM Systems Journal **44** (2005), no. 2, 259–278.

[5] Free Software Foundation, http://gcc.gnu.org/onlinedocs/, *Gcc internals manual*.

[6] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and Francois Bodin, *MILEPOST GCC: Machine Learning Based Research Compiler*, GCC Developers' Summit (2008), 1–13.

[7] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley Professional, 1994.

[8] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 2006.

[9] International Telecommunication Union, http://www.itu.int/, *International mobile telecommunications-2000 standard*.

[10] Georgia Kouveli, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris, *Exploring the Benefits of Randomized Instruction Scheduling*, GROW (2011).

[11] Wing-Ning Li, Zhichun Xiao, and Gordon Beavers, *On Computing the Number of Topological Orderings of a Directed Acyclic Graph*, Congressus Numerantium **174** (2005), 143–159.

[12] Vladimir N. Makarov, *The Integrated Register Allocator for GCC*, GCC Developers' Summit (2007), 77–90.

[13] Abid M. Malik, Tyrel Russell, Michael Chase, and Peter van Beek, *Learning Heuristics for Basic Block Instruction Scheduling*, Journal of Heuristics **14** (1981), no. 6, 549–569.

[14] Thomas Muller, *Employing Finite Automata for Resource Scheduling*, Proceedings of the 26th Annual International Symposium on Microarchitecture (1993), 12–20.

[15] D. Novillo, *Tree SSA: A New Optimization Infrastructure for GCC*, GCC Developers' Summit (2003), 181–193.

[16] David A. Patterson and Carlo H. Sequin, *RISC I: A Reduced Instruction Set VLSI Computer*, Proceedings of the 8th Annual Symposium on Computer Architecture (1981), 443–457.

[17] Todd A. Proebsting and Christopher W. Fraser, *Detecting Pipeline Structural Hazards Quickly*, Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (1994), 280–286.

[18] Stuart Russell and Peter Norvig, *Artificial Intelligence, A Modern Approach*, Prentice Hall, 2009.

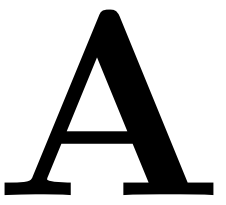
[19] Philip J. Schielke, *Stochastic Instruction Scheduling*, Ph.D. thesis, Rice University, 2000.

[20] Y.N. Srikant and Priti Shankar (eds.), *The Compiler Design Handbook: Optimizations & Machine Code Generation*, CRC Press, 2002.

[21] Roel Trienekens, *Porting the GCC Compiler to a VLIW Vector Processor*, Master's thesis, Delft University of Technology, 2009.

[22] Alex Turjan, Dmitry Cheresiz, Claudiu Zissulescu, and Wim Kloosterhuis, *The GCC Port for the EVP Architecture*, ST-Ericsson DSP Innovation Center, 2012.

Appendix A: Benchmarks for "kernels-321"



Group	Test Case	LS	Iterative-LS	Improved	Geomean
1	golay_manoverlay	1146.33	575.00	49.84%	19.234%
	cowc_fxp_kernel	3135.00	2082.00	33.59%	
	golay_manpipe1	1157.33	897.33	22.47%	
	libEvpMatInvSqrt	234.00	182.00	22.22%	
	LTE_FHT32pt	123.00	100.00	18.70%	
	LTE_initial_FHT32pt	123.00	100.00	18.70%	
	demuxb_2	82.00	68.00	17.07%	
	demuxb_2_val8	84.00	70.00	16.67%	
	fht_s4_1	72.00	60.00	16.67%	
	fht_s4_2	72.00	60.00	16.67%	
	LTE_periodic_InputPermute	302.00	254.00	15.89%	
	fht_s4_1_limit	71.00	60.00	15.49%	
	fht_s4_2_limit	71.00	60.00	15.49%	
	libEvpTranspose	703.00	595.00	15.36%	
	t3g_transpose	758.00	650.00	14.25%	
	demux_2	67.00	58.00	13.43%	
	fft_block_4096	1953.00	1697.00	13.11%	
	fft_block_2048	993.00	865.00	12.89%	
	fft_block_1024	513.00	449.00	12.48%	
	fft_64_selfsort_note	450.00	394.00	12.44%	
2	fft_64_selfsort_nsat	450.00	394.00	12.44%	10.535%
	fft_block_512	273.00	241.00	11.72%	
	libEvpFht32HP_gcc	45.00	40.00	11.11%	
	fft_2048	3759.00	3342.00	11.09%	
	fft_1024	1773.00	1579.00	10.94%	
	cowc_flp_kernel	1965.00	1751.00	10.89%	
	h264transform_4_fa_mp	178.00	159.00	10.67%	
	fft_64_2_pipe	462.00	413.00	10.61%	
	fft_64_sat_2_pipe	462.00	413.00	10.61%	
	fft_4096	7892.00	7058.00	10.57%	
	fft_block_256	153.00	137.00	10.46%	
	fft_64_1	477.00	428.00	10.27%	
	fft_64_2	477.00	428.00	10.27%	
	fft_64_sat_2	477.00	428.00	10.27%	
	libEvpFft128_div2	139.00	125.00	10.07%	
	fft_4096_btail_vi	1311.00	1181.00	9.92%	
	fft_4096_btail_ii	1318.00	1190.00	9.71%	
	fft_1024_tail_vi	351.00	317.00	9.69%	
	libEvpFft64_div2	62.00	56.00	9.68%	
	libEvpFft256	311.00	281.00	9.65%	

Table A.1: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
3	libEvpFft256_sat	311.00	281.00	9.65%	8.643%
	fft_64_1_pipe	462.00	418.00	9.52%	
	fft_2048_btail_ii	678.00	614.00	9.44%	
	fft_512_btail_vi	191.00	173.00	9.42%	
	fft_2048_btail_vvi	1057.00	960.00	9.18%	
	libEvpDecimateBy3	997.00	906.00	9.13%	
	fft_1024_btail_ii	358.00	326.00	8.94%	
	demux_3	68.00	62.00	8.82%	
	fft_128_block_vvv	93.00	85.00	8.60%	
	libEvpFft128HP	444.00	406.00	8.56%	
	libEvpFft256_div2	328.00	300.00	8.54%	
	fft_128_btail_vi	71.00	65.00	8.45%	
	LTE_MetricsScalar	1919.00	1758.00	8.39%	
	LTE_initial_MetricsScalar	1919.00	1758.00	8.39%	
	fft_512	893.00	819.00	8.29%	
	fft_512_btail_ii	198.00	182.00	8.08%	
	fft_256_btail_vvi	161.00	148.00	8.07%	
	fft_256	464.00	427.00	7.97%	
	libEvpLog10Fl_v	13.00	12.00	7.69%	
	libEvpLogEFl_v	13.00	12.00	7.69%	
4	mc_16xN_b	205.00	190.00	7.32%	6.299%
	libEvpFftDynScal_8192	17801.00	16516.00	7.22%	
	libEvpFftDynScal_sat_8192	17801.00	16516.00	7.22%	
	fft_256_btail_ii	118.00	110.00	6.78%	
	LTE_CollectSsigSpectra	612.00	571.00	6.70%	
	fft_note_idx	422.00	394.00	6.64%	
	fft_64_selfsort_nidx	422.00	394.00	6.64%	
	fft_128	274.00	256.00	6.57%	
	huffdecode_opt1	1476.00	1382.00	6.37%	
	fft2_inter_4	126.00	118.00	6.35%	
	libEvpFft128	127.00	119.00	6.30%	
	libEvpFft128_sat	127.00	119.00	6.30%	
	LTE_periodic_FFT128	127.00	119.00	6.30%	
	libEvpLog10Fxp	17.00	16.00	5.88%	
	fft_64_selfsort_swp1	404.00	381.00	5.69%	
	fft_64_selfsort_swp2	404.00	381.00	5.69%	
	libEvpAtan2Fl	18.00	17.00	5.56%	
	LTE_FFT128	127.00	120.00	5.51%	
	LTE_initial_FFT128	127.00	120.00	5.51%	
	libEvpFft32	37.00	35.00	5.41%	

Table A.2: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
5	libEvpFft32Fl	37.00	35.00	5.41%	4.875%
	fxpfft1	57956.00	54884.00	5.30%	
	fxpfft2	57956.00	54884.00	5.30%	
	fxpfft3	57956.00	54884.00	5.30%	
	Alg_InterpolatedChest_2	191.00	181.00	5.24%	
	LTE_DeinterleaveInplace	39.00	37.00	5.13%	
	LTE_initial_DeinterleaveInplace	39.00	37.00	5.13%	
	LTE_periodic_DeinterleaveInplace	39.00	37.00	5.13%	
	btail_ii	78.00	74.00	5.13%	
	fft_64_selfsort_2	411.00	390.00	5.11%	
	btail_vv	1313.00	1249.00	4.87%	
	libEvpFht32HP	42.00	40.00	4.76%	
	Alg_MatFilt	3109.00	2964.00	4.66%	
	BDTI_OFDM_IQ_SLICER_1	1400.00	1336.00	4.57%	
	libEvpFht16HP_gcc	22.00	21.00	4.55%	
	BDTI_OFDM_IQ_SLICER_2	1450.00	1386.00	4.41%	
	fft4bfy_1_stepreg	91.00	87.00	4.40%	
	fft4bfy_2_bcst	91.00	87.00	4.40%	
	fft4bfy_3_ptrupd	92.00	88.00	4.35%	
	fft4bfy_4_aliases	92.00	88.00	4.35%	
6	h264transform_2	260.00	249.00	4.23%	3.602%
	LTE_GetPsigChest	48.00	46.00	4.17%	
	LTE_initial_GetEquSsigSpectrum	48.00	46.00	4.17%	
	LTE_initial_GetPsigChest	48.00	46.00	4.17%	
	LTE_periodic_EquSsigSpectrum	48.00	46.00	4.17%	
	LTE_periodic_GetPsigChest	48.00	46.00	4.17%	
	btail_vv	193.00	185.00	4.15%	
	LTE_initial_1	23883.00	22930.00	3.99%	
	libEvpFft64	54.00	52.00	3.70%	
	libEvpFft64_sat	54.00	52.00	3.70%	
	BDTI_OFDM_IQ_SLICER_3	2667.00	2571.00	3.60%	
	fft_64_selfsort_sat	404.00	390.00	3.47%	
	tgolay_manoverlay2_golay	1062.00	1025.33	3.45%	
	h264transform_1	325.00	314.00	3.38%	
	libEvpCxNorm	30.00	29.00	3.33%	
	LTE_PsigChest	127.00	123.00	3.15%	
	golay_manpipe2	1352.67	1312.00	3.01%	
	BDTI_OFDM_IQ_SLICER_4	1707.00	1659.00	2.81%	
	LTE_periodic_1	5162.00	5024.00	2.67%	
	W_AP_PFU_CCES_HS_1	79.00	77.00	2.53%	

Table A.3: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
7	W_AP_PFU_CCES_HS_2	79.00	77.00	2.53%	1.991%
	pccfAcq_1	42.00	41.00	2.38%	
	pccfAcq_reuse	42.00	41.00	2.38%	
	pccfAcq_2	42.00	41.00	2.38%	
	multi_loops	87.00	85.00	2.30%	
	Alg_InterpolatedChest_1	131.00	128.00	2.29%	
	fft_64_selfsort_3	397.00	388.00	2.27%	
	libEvpInterpolateBy5	1251.00	1226.00	2.00%	
	Alg_InterferenceHs_2	258.00	253.00	1.94%	
	LTE_GetEquSpectrum	52.00	51.00	1.92%	
	LTE_periodic_SsigEqu	54.00	53.00	1.85%	
	LTE_periodic_GetAvgEquSpectrum	55.00	54.00	1.82%	
	libEvpDecimateBy5	1465.00	1439.00	1.77%	
	Alg_InterferenceHs_1	283.00	278.00	1.77%	
	LTE_Derotate	170.00	167.00	1.76%	
	LTE_Permute	172.00	169.00	1.74%	
	LTE_initial_Permute	172.00	169.00	1.74%	
	LTE_periodic_CollectSsigSpectra	58.00	57.00	1.72%	
	LTE_initial_SsigEqu	59.00	58.00	1.69%	
	libEvpInterpolateBy3	784.00	772.00	1.53%	
8	libEvpInterpolateBy3_gcc	784.00	772.00	1.53%	1.145%
	LTE_periodic_TestHypothesis	66.00	65.00	1.52%	
	libEvpChol	44911.00	44274.00	1.42%	
	pccf1_main	71.00	70.00	1.41%	
	pccf1_main_resuse	71.00	70.00	1.41%	
	pccf2_main	71.00	70.00	1.41%	
	libEvpTranspose_gcc	612.00	604.00	1.31%	
	LTE_EvalHypotheses	78.00	77.00	1.28%	
	t3g_les1	5549.00	5485.00	1.15%	
	tfft4bfy_6_muldep_2	88.00	87.00	1.14%	
	cck5511Enc_manual_swp	91.00	90.00	1.10%	
	libEvpLes1Fl	586.00	580.00	1.02%	
	LTE_periodic_FHT32pt	100.00	99.00	1.00%	
	fft2_inter_5	101.00	100.00	0.99%	
	cces_hs_W_AP_PFU_CCES_HS	816.00	808.00	0.98%	
	libEvpInterpolateBy5	1238.00	1226.00	0.97%	
	libEvpInvChol	161819.00	160370.00	0.90%	
	libEvpMatrixMultiplyFl	30017.00	29777.00	0.80%	
	t3g_les2	5602.00	5558.00	0.79%	
	libEvpLes	116882.00	115976.00	0.78%	

Table A.4: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
9	libEvpLes1Fxp	789.00	783.00	0.76%	0.165%
	libEvpMatrixMultiply	32908.00	32668.00	0.73%	
	W_AP_PFU_CCES_HS	716.00	712.00	0.56%	
	LTE_initial_AverageSpectra	647.00	645.00	0.31%	
	LTE_periodic_OutputPermute	372.00	371.00	0.27%	
	libEvpCholFxp	10084.00	10065.00	0.19%	
	libEvpCholFxp_gcc	10102.00	10083.00	0.19%	
	LTE_initial_CollectSsigSpectra	615.00	614.00	0.16%	
	tvpressure_do_0	1045.00	1044.00	0.10%	
	t3g_cholNxN	7374.00	7373.00	0.01%	
	libEvpCholRDRFl	8301.00	8300.00	0.01%	
	read_write_I_c_4M_to_4M	8.00	8.00	0.00%	
	read_write_I_p_4M_to_4M	8.00	8.00	0.00%	
	write_write_I_M_M	8.00	8.00	0.00%	
	read_write_c_4M_to_4M	8.00	8.00	0.00%	
	read_write_c_4M_to_4M_x	8.00	8.00	0.00%	
	read_write_p_4M_to_4M	8.00	8.00	0.00%	
	write_write_M_M	8.00	8.00	0.00%	
	BDTI_OFDM_FIR_1	8303.00	8303.00	0.00%	
	BDTI_OFDM_FIR_2	8302.00	8302.00	0.00%	
10	BDTI_OFDM_IQ_Demod	42.00	42.00	0.00%	0.000%
	BDTI_OFDM_Viterbi	11181.00	11181.00	0.00%	
	libEvpAtan2	29.00	29.00	0.00%	
	libEvpAtan2Fl_v	17.00	17.00	0.00%	
	libEvpDecimateBy2	693.00	693.00	0.00%	
	libEvpFht16HP	21.00	21.00	0.00%	
	libEvpInterpolateBy2	576.00	576.00	0.00%	
	libEvpInterpolateBy2_gcc	576.00	576.00	0.00%	
	libEvpInvSqrt32ToFloat	12.00	12.00	0.00%	
	libEvpInvSqrt32ToFloat_v	18.00	18.00	0.00%	
	libEvpLes2Fl	973.00	973.00	0.00%	
	libEvpLes2Fxp	1749.00	1749.00	0.00%	
	libEvpLog10Fl	12.00	12.00	0.00%	
	libEvpLog2Fl	12.00	12.00	0.00%	
	libEvpLog2Fl_v	11.00	11.00	0.00%	
	libEvpLogEFl	12.00	12.00	0.00%	
	libEvpLog10Fxp_v	20.00	20.00	0.00%	
	libEvpLog2Fxp	15.00	15.00	0.00%	
	libEvpLog2Fxp_v	20.00	20.00	0.00%	
	libEvpLogEFxp	15.00	15.00	0.00%	

Table A.5: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
11	libEvpLogEFxp_v	20.00	20.00	0.00%	0.000%
	libEvpCos_v	16.00	16.00	0.00%	
	libEvpSin_v	15.00	15.00	0.00%	
	libEvpSqrt32	10.00	10.00	0.00%	
	libEvpSqrt32_v	19.00	19.00	0.00%	
	libEvpSqrtFl	11.00	11.00	0.00%	
	libEvpSqrtFl_v	11.00	11.00	0.00%	
	fxpAutoCorrelation1	4958568.00	4958568.00	0.00%	
	fxpAutoCorrelation2	228232.00	228232.00	0.00%	
	fxpAutoCorrelation3	1680.00	1680.00	0.00%	
	fxpAutoCorrelation4	218104.00	218104.00	0.00%	
	convolutionalEncode1	124976.00	124976.00	0.00%	
	convolutionalEncode2	82984.00	82984.00	0.00%	
	convolutionalEncode3	124976.00	124976.00	0.00%	
	fxpBitAllocation1	159364.00	159364.00	0.00%	
	fxpBitAllocation2	41833.00	41833.00	0.00%	
	fxpBitAllocation3	552644.00	552644.00	0.00%	
	idct_int32	449.00	449.00	0.00%	
	ViterbiDecoderIS136_0	4951.00	4951.00	0.00%	
	ViterbiDecoderIS136_1	235604.00	235604.00	0.00%	
12	ViterbiDecoderIS136_2	235604.00	235604.00	0.00%	0.000%
	ViterbiDecoderIS136_3	235604.00	235604.00	0.00%	
	ViterbiDecoderIS136_4	235604.00	235604.00	0.00%	
	load_to_ofs	5130.00	5130.00	0.00%	
	move_no_valu	10.00	10.00	0.00%	
	vmove_no_vshu	10.00	10.00	0.00%	
	vmove_valu	10.00	10.00	0.00%	
	vmove_vlsu	10.00	10.00	0.00%	
	vmove_vshu	10.00	10.00	0.00%	
	pressure_do_1	1047.00	1047.00	0.00%	
	pressure_do_2	1047.00	1047.00	0.00%	
	pressure_doi_1	1045.00	1045.00	0.00%	
	pressure_doi_2	1045.00	1045.00	0.00%	
	pressure_doi_3	1045.00	1045.00	0.00%	
	vpressure_doi	1044.00	1044.00	0.00%	
	LTE_CalcCos	16.00	16.00	0.00%	
	LTE_CalcSin	15.00	15.00	0.00%	
	LTE	76.00	76.00	0.00%	
	LTE_AverageSpectra	508.00	508.00	0.00%	
	LTE_CalcMetrics	97.00	97.00	0.00%	

Table A.6: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
13	LTE_DescCSeq_1	34.00	34.00	0.00%	0.000%
	LTE_DescZSeq_2	93.00	93.00	0.00%	
	LTE_GenDerotSeq	126.00	126.00	0.00%	
	LTE_SsigEqu	53.00	53.00	0.00%	
	LTE_TestHypothesis	48.00	48.00	0.00%	
	LTE_Update	37.25	37.25	0.00%	
	LTE_main	268.00	268.00	0.00%	
	LTE_initial_CalcCos	16.00	16.00	0.00%	
	LTE_initial_CalcSin	15.00	15.00	0.00%	
	LTE_initial_2	84.00	84.00	0.00%	
	LTE_initial_CalcMetrics	89.00	89.00	0.00%	
	LTE_initial_Derotate	157.00	157.00	0.00%	
	LTE_initial_DescCSeq_1	34.00	34.00	0.00%	
	LTE_initial_DescZSeq_2	93.00	93.00	0.00%	
	LTE_initial_EvalHypotheses	88.00	88.00	0.00%	
	LTE_initial_GenDerotSeq	118.00	118.00	0.00%	
	LTE_initial_PsigChest	81.00	81.00	0.00%	
	LTE_initial_TestHypothesis	61.00	61.00	0.00%	
	LTE_initial_Update	26.00	26.00	0.00%	
	LTE_periodic_2	48.00	48.00	0.00%	
14	LTE_periodic_AvgSsigSpectra	92.00	92.00	0.00%	0.000%
	LTE_periodic_CalcMetrics	66.00	66.00	0.00%	
	LTE_periodic_DescCSeq_1	34.00	34.00	0.00%	
	LTE_periodic_DescZSeq_2	93.00	93.00	0.00%	
	LTE_periodic_EvalCPHypothesis	43.00	43.00	0.00%	
	LTE_periodic_EvalSFHypothesis	29.00	29.00	0.00%	
	LTE_periodic_MetricsVector	85.00	85.00	0.00%	
	LTE_periodic_PsigChest	81.00	81.00	0.00%	
	LTE_periodic_Unpack	86.00	86.00	0.00%	
	cces_hs_Alg_ChEst	72.00	72.00	0.00%	
	cces_hs_Alg_ChSamp	156.00	156.00	0.00%	
	cces_hs_Util_Iir	35.00	35.00	0.00%	
	cces_Alg_ChEst	72.00	72.00	0.00%	
	cces_Alg_ChSamp	141.00	141.00	0.00%	
	cces_Util_Iir	35.00	35.00	0.00%	
	bitscount_2	85.00	85.00	0.00%	
	cck5511Enc	146.00	146.00	0.00%	
	cck5511Enc_main	2088.00	2088.00	0.00%	
	dct_1	141.00	141.00	0.00%	
	dct_2	133.00	133.00	0.00%	

Table A.7: Average execution cycle counts for "kernels-321": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
15	dct_tdct_1	932.00	932.00	0.00%	0.000%
	dct_transform_1	547.00	547.00	0.00%	
	dct_tdct_1	948.00	948.00	0.00%	
	dct_transform_2	547.00	547.00	0.00%	
	demux_0	113.00	113.00	0.00%	
	one_demux_0	92.00	92.00	0.00%	
	demux_1	71.00	71.00	0.00%	
	demuxb_1	79.00	79.00	0.00%	
	fft2_inter	174.00	174.00	0.00%	
	fft2_inter_1	178.00	178.00	0.00%	
	fft2_inter_2	174.00	174.00	0.00%	
	fft2_inter_3	174.00	174.00	0.00%	
	fft2_inter_4	176.00	176.00	0.00%	
	fft2_inter_5	169.00	169.00	0.00%	
	fft2_inter_6	169.00	169.00	0.00%	
	fft2_inter_7	136.00	136.00	0.00%	
	fft4bfy_0	86.00	86.00	0.00%	
	fft4bfy_5_muldep_1	103.00	103.00	0.00%	
	fft4bfy_5_muldep_1	103.00	103.00	0.00%	
16	fht_0_orig	168.00	168.00	0.00%	0.000%
	fht_1_aliases	163.00	163.00	0.00%	
	fht_2_notemp	163.00	163.00	0.00%	
	fht_3_lessreg	163.00	163.00	0.00%	
	fht_lessreg_exp	163.00	163.00	0.00%	
	fht_s4_0	153.00	153.00	0.00%	
	fht_1_aliases	142.00	142.00	0.00%	
	fir_1	20944.00	20944.00	0.00%	
	fir_2	11600.00	11600.00	0.00%	
	fir_3	11284.00	11284.00	0.00%	
	fir_scalar_1	334863.00	334863.00	0.00%	
	golay_aliases_golay	1204.00	1204.00	0.00%	
	golay_manoverlay3_golay	980.67	980.67	0.00%	
	golay_noimm_golay	1207.00	1207.00	0.00%	
	golay_orig_golay	1287.33	1287.33	0.00%	
	th264transform	230.00	230.00	0.00%	
	huffdecode	2063.00	2063.00	0.00%	
	one_parity	13.00	13.00	0.00%	
	parity	110.00	110.00	0.00%	
	sqrt8_main	96.00	96.00	0.00%	
	sqrt8	18.00	18.00	0.00%	
	vmfill	13.00	13.00	0.00%	

Table A.8: Average execution cycle counts for "kernels-321": LS vs. ILS.

Appendix B: Benchmarks for "kernels-81"

B

Group	Test Case	LS	Iterative-LS	Improved	Geomean
1	convolutionalEncode1	97318.00	90150.00	7.37%	5.966%
	convolutionalEncode3	97318.00	90150.00	7.37%	
	cowc_fxp_kernel	2108.00	1956.00	7.21%	
	libEvpFft256	302.00	283.00	6.29%	
	libEvpFft256_sat	302.00	283.00	6.29%	
	libEvpFft256_div2	318.00	298.00	6.29%	
	convolutionalEncode2	66594.00	62498.00	6.15%	
	libEvpFht32HP	40.00	38.00	5.00%	
	libEvpFft128_div2	129.00	124.00	3.88%	
	cowc_flp_kernel	1688.00	1625.00	3.73%	
2	libEvpFftDynScal	16778.00	16262.00	3.08%	1.780%
	libEvpFftDynScal_sat	16778.00	16262.00	3.08%	
	libEvpMatInvSqrt	133.00	129.00	3.01%	
	libEvpFft128HP	408.00	401.00	1.72%	
	libEvpDecimator	919.00	906.00	1.41%	
	idct32	411.00	406.00	1.22%	
	fft_1024	1467.00	1450.00	1.16%	
	fft_2048	3018.00	2985.00	1.09%	
	fft_4096	6546.00	6480.00	1.01%	
	libEvpInvChol	150156.00	148658.00	1.00%	
3	libEvpFft128	122.00	121.00	0.82%	0.448%
	libEvpFft128_sat	122.00	121.00	0.82%	
	libEvpChol	43282.00	42929.00	0.82%	
	libEvpMatrixMultiplyFl	30016.00	29776.00	0.80%	
	fxpAutoCorrelation3	1048.00	1040.00	0.76%	
	libEvpCholFxp	8614.00	8581.00	0.38%	
	fxpAutoCorrelation4	124952.00	124920.00	0.03%	
	viterb01	212054.00	212018.00	0.02%	
	t3g_cholNxN	6104.00	6103.00	0.02%	
	libEvpCholRDRFl	7521.00	7520.00	0.01%	
4	fxpAutoCorrelation2	130584.00	130568.00	0.01%	0.001%
	fxpAutoCorrelation1	2833944.00	2833896.00	0.00%	
	libEvpLes	103250.00	103249.00	0.00%	
	libEvpAtan2	28.00	28.00	0.00%	
	libEvpAtan2Fl	15.00	15.00	0.00%	
	libEvpAtan2Fl_v	16.00	16.00	0.00%	
	libEvpCxNorm	29.00	29.00	0.00%	
	libEvpDecimateBy2	692.00	692.00	0.00%	
	libEvpDecimateBy5	1465.00	1465.00	0.00%	
	libEvpFft32	34.00	34.00	0.00%	

Table B.1: Average execution cycle counts for "kernels-81": LS vs. ILS.

Group	Test Case	LS	Iterative-LS	Improved	Geomean
5	libEvpFft32Fl	35.00	35.00	0.00%	0.000%
	libEvpFft64	52.00	52.00	0.00%	
	libEvpFft64_div2	57.00	57.00	0.00%	
	libEvpFft64_sat	52.00	52.00	0.00%	
	libEvpFht16HP	21.00	21.00	0.00%	
	libEvpInterpolateBy2	576.00	576.00	0.00%	
	libEvpInterpolateBy3	760.00	760.00	0.00%	
	libEvpInterpolateBy5	1226.00	1226.00	0.00%	
	libEvpInvSqrt	12.00	12.00	0.00%	
	libEvpInvSqrt_v	17.00	17.00	0.00%	
6	libEvpLesFl	580.00	580.00	0.00%	0.000%
	libEvpLesFl_v	960.00	960.00	0.00%	
	libEvpLesFxp	783.00	783.00	0.00%	
	libEvpLesFxp_v	1449.00	1449.00	0.00%	
	libEvpLogFl	12.00	12.00	0.00%	
	libEvpLogFl_v	12.00	12.00	0.00%	
	libEvpLogFl	12.00	12.00	0.00%	
	libEvpLogFl_v	11.00	11.00	0.00%	
	libEvpLogFl	12.00	12.00	0.00%	
	libEvpLogFl_v	12.00	12.00	0.00%	
7	libEvpLogFxp	16.00	16.00	0.00%	0.000%
	libEvpLogFxp_v	20.00	20.00	0.00%	
	libEvpLogFxp	14.00	14.00	0.00%	
	libEvpLogFxp_v	19.00	19.00	0.00%	
	libEvpLogFxp	14.00	14.00	0.00%	
	libEvpLogFxp_v	19.00	19.00	0.00%	
	libEvpMatrixMultiply	31132.00	31132.00	0.00%	
	libEvpSin_v	16.00	16.00	0.00%	
	libEvpSin_v	15.00	15.00	0.00%	
	libEvpSqrt	10.00	10.00	0.00%	
8	libEvpSqrt_v	17.00	17.00	0.00%	0.000%
	libEvpSqrtFl	11.00	11.00	0.00%	
	libEvpSqrtFl_v	11.00	11.00	0.00%	
	libEvpTranspose	581.00	581.00	0.00%	
	fxpBitAllocation1	107995.00	107995.00	0.00%	
	fxpBitAllocation2	28573.00	28573.00	0.00%	
	fxpBitAllocation3	374587.00	374587.00	0.00%	
	fxpfft	41823.00	41823.00	0.00%	
	t3g_les1	5485.00	5485.00	0.00%	
	t3g_les2	5555.00	5555.00	0.00%	
	t3g_transpose	495.00	495.00	0.00%	

Table B.2: Average execution cycle counts for "kernels-81": LS vs. ILS.

Appendix C: Branch Promotion Benchmark

C

In this section, the measured performance of the compiler is compared before and after adding the branch promotion feature. The experiment was performed on "kernels-321" benchmark. In Table C.1, only the affected test cases are listed. The branch promotion led to both improvements and degradations. However, as it can be observed in Figure C.1, the number and magnitude of improvements exceeds that of degradations.

The branch promotion approach gives higher priority to the predecessors of a branch so that they can be scheduled in the early cycles of the basic block. If these promoted instructions do not postpone the critical instructions, then there is a possibility that the number of cycles will be reduced; otherwise, the result can be a longer schedule.

Test Case	**Without Branch Promotion**	**With Branch Promotion**	**Improved**
LTE_GetPsigChest	47.00	48.00	-2.13%
LTE_periodic_GetPsigChest	47.00	48.00	-2.13%
LTE_initial_GetEquSsigSpectrum	47.00	48.00	-2.13%
LTE_periodic_EquSsigSpectrum	47.00	48.00	-2.13%
LTE_initial_GetPsigChest	47.00	48.00	-2.13%
LTE_GetEquSpectrum	51.00	52.00	-1.96%
LTE_FHT32pt	121.00	123.00	-1.65%
LTE_initial_FHT32pt	121.00	123.00	-1.65%
fft2_inter_5	100.00	101.00	-1.00%
libEvpTranspose_gcc	607.00	612.00	-0.82%
Alg_InterferenceHs_2	256.00	258.00	-0.78%
fft_512_btail_vi	192.00	191.00	0.52%
cces_hs_Alg_ChSamp	157.00	156.00	0.64%
cces_Alg_ChSamp	142.00	141.00	0.70%
Alg_InterpolatedChest_1	132.00	131.00	0.76%
W_AP_PFU_CCES_HS	722.00	716.00	0.83%
cces_hs_W_AP_PFU_CCES_HS	824.00	816.00	0.97%
fft2_inter_1	180.00	178.00	1.11%
libEvpLes2Fl	984.00	973.00	1.12%
convolutionalEncode1	126512.00	124976.00	1.21%
convolutionalEncode3	126512.00	124976.00	1.21%
fft_128_btail_vi	72.00	71.00	1.39%
convolutionalEncode2	84520.00	82984.00	1.82%
LTE_initial_1	24463.00	23883.00	2.37%
W_AP_PFU_CCES_HS_1	82.00	79.00	3.66%
W_AP_PFU_CCES_HS_2	82.00	79.00	3.66%
huffdecode_opt1	1536.00	1476.00	3.91%
huffdecode	2155.00	2063.00	4.27%
LTE_initial_MetricsScalar	2072.00	1919.00	7.38%
LTE_MetricsScalar	2072.00	1919.00	7.38%
cces_hs_Util_Iir	38.00	35.00	7.89%
cces_Util_Iir	38.00	35.00	7.89%

Table C.1: Average execution times for “kernels-321” affected by branch promotion.

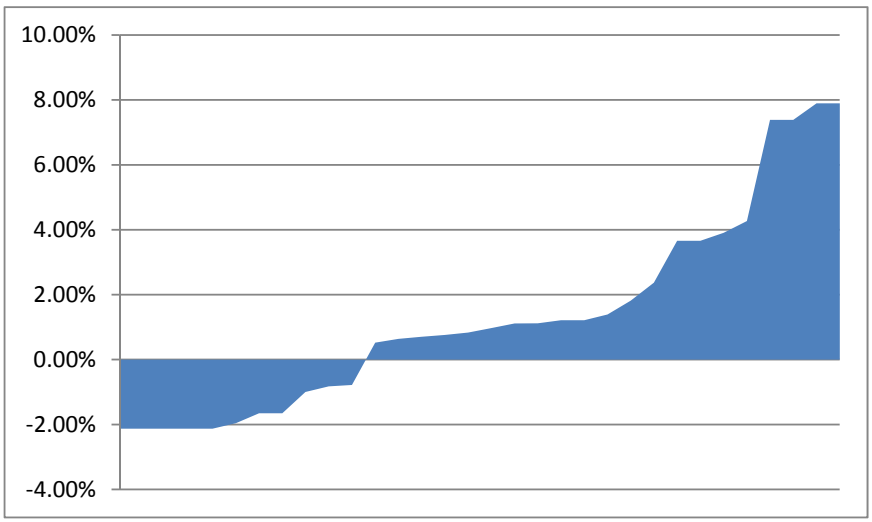


Figure C.1: The effect of branch promotion on “kernels-321”.

List of Acronyms

ACU	Address Computation Unit
BFS	Breadth First Search ... 2
CGU	Code Generation Unit ... 16
COWC	COmbining Weight Computation ... vi
CPL	Critical Path Length ... 19
DAG	Directed Acyclic Graph ... 3
DCE	Dead Code Elimination ... 6
DDG	Data Dependence Graph ... v
DFA	Deterministic Finite Automaton ... 7
DFS	Depth First Search ... 2
DSP	Digital Signal Processor ... i
EVP	Embedded Vector Processor ... i
GCC	GNU Compiler Collection ... i
GSM	Groupe Spcial Mobile
ILP	Instruction Level Parallelism ... i
ILS	Iterative List Scheduler ... xi
IRA	Integrated Register Allocator ... 6
ISA	Instruction Set Architecture ... 5
IVU	Intra Vector Unit ... 16
LS	List Scheduler ... xi
LTE	Long-Term Evolution
modem	MOdulator-DEModulator
NOP	No Operation (instruction)
PALU	Predicated Arithmetic Logic Unit
PCU	Program Control Unit
PRNG	Pseudo-Random Number Generator ... 21
RISC	Reduced Instruction Set Computer ... 1
RTL	Register Transfer Language ... vii
RTX	RTL Expression ... 45
SALU	Scalar Arithmetic Logic Unit
SDCU	Scalar Data Computation Unit
SLSU	Scalar Load/Store Unit ... 16

SMAC Scalar Multiply/Accumulate 16
SSA Static Single Assignment 5
UID Unique Instruction Identifier 9
UMTS Universal Mobile Telecommunications System 16
VALU Vector Arithmetic Logic Unit 16
VMALU Vector Mask Arithmetic Logic Unit 16
VDCU Vector Data Computation Unit
VLIW Very Long Instruction Word i
VLSU Vector Load/Store Unit 16
VMAC Vector Multiply/Accumulate 16
VMALU Vector Mask Arithmetic Logic Unit 16
VSHU Vector Shuffle Unit 16
WCDMA Wideband Code Division Multiple Access