Porting GCC to Exposed Pipeline VLIW Processors

Alexandru Turjan
ST-Ericsson
alex.turjan@stericsson.com

Dmitry Cheresiz
ST-Ericsson
dmitry.cheresiz@stericsson.com

Roel Trienekens
Delft University of Technology
roel.trienekens@gmail.com

Abstract

EVP and TriMedia are embedded application processors targeted at mobile communication and multimedia domains. Both architectures originate from Philips Semiconductors and are currently developed by ST-Ericsson and NXP Semiconductors, respectively. Both processors have a VLIW architecture with an exposed pipeline. Such architectures impose different requirements on a compiler than the majority of existing GCC targets, which are scalar or superscalar machines with interlocked pipelines. First, the exposed pipeline organization requires a compiler to schedule operations such that all data and resource hazards are avoided. Second, a compiler for a VLIW machine has to provide stronger capabilities for discovering and exposing the instruction level parallelism (ILP), as it can not rely on the hardware ILP mechanisms employed in superscalar processors. We have ported GCC to EVP and TriMedia and provided extensions to support code generation for an exposed pipeline VLIW. To increase the amount of exploitable ILP, we have also enhanced the current GCC mechanisms such as loop unrolling and the alias analysis. The ports were benchmarked against the existing production compilers and encouraging results in terms of cycle counts and code size have been achieved.

1 Introduction

ST-Ericsson and NXP Semiconductors develop embedded *systems-on-chip (SoCs)* providing signal or media processing functionality for the products like mobile phones, TVs and set-top boxes. Such an SoC consists of several components. The *host processor* executes the control tasks such as running embedded OS, providing user interface, etc. Typically, the host processor is a general purpose processor (GPP) based on an existing architecture, such as ARM. It controls the rest of the system and dispatches the tasks to one or several *application processors (APs)* and, optionally, hardware accelerators. An application processor provides the core functionality of an SoC and runs the most compute intensive tasks. The *Embedded Vector Processor (EVP)* [1, 2] and the members of *TriMedia* processor family [3, 4] are examples of such processors. An application processor is required to provide very high performance/power ratio on a limited set of algorithms from a certain media or signal-processing domain. For example, for the inner receiver of the LTE standard, at which EVP is targeted, an application processor should provide 13 GOPS with a power budget of less than 400 mW.

The algorithms mapped on APs exhibit significant amounts parallelism which facilitate the achievement of performance/power targets. *Data-level parallelism (DLP)* is commonly exploited by means of vector instructions. For example, EVP uses dedicated 256-bit vector registers, while TriMedia provides vector operations on existing 32-bit scalar registers. We remark that contemporary GPPs take the same approach in exploiting the DLP, introducing short-vector instructions, e.g. AltiVec and SSE ISA extensions. For exploiting the *instruction-level parallelism (ILP)*, however, the approaches used in GPPs and in embedded APs differ significantly. Due to need for legacy code support, binary code compatibility is often a must for a GPP. Therefore, such a processor accepts sequential code and exploits the ILP by means of hardware mechanisms which discover the independent instructions in the program flow and issue/execute them in parallel. Compiler for such a processor plays a limited role in exposing the ILP, leaving the main responsibility to the hardware.

This approach would be too costly for an embedded pro-

cessor in terms of area and power required for the hardware ILP mechanisms. At the same time the algorithms in the embedded domain exhibit more regularity and allow deeper static analysis, while the binary compatibility is not necessarily required. This motivated wide adoption of the *VLIW* concept for exploiting the ILP in embedded processors. In a *VLIW* system, the compiler is responsible to discover independent operations and pack them into a *Very Long Instruction Word (VLIW)*. EVP and TriMedia addressed in this paper have a VLIW architecture.

Currently, EVP and TriMedia are supported by tool chains based on proprietary compilers [5, 6]. However, rapid development of GCC and introduction of new features such as auto-vectorization motivated us to investigate its capabilities with respect to these processors. This has lead to the setup of two study projects in which we ported GCC to EVP and TriMedia and investigated its advantages and limitations, particularly with respect to exposed VLIW architecture support. In this paper we present the experience gained during these projects, the techniques we employed to target GCC, and some of the achieved results. The goal of the projects has been to generate correct and efficient VLIW code using GCC, and, desirably, to achieve the level of performance comparable with our existing production compilers.

The main steps in VLIW compilation are *sequential code generation* and *scheduling*. The first step consists of converting the input code into a sequence of operations supported by the target. It corresponds to the usual compilation flow for scalar or superscalar processors and is performed in the same way. During the second step, the individual operations are packed together into VLIW instructions such that the data dependencies and resource constraints are satisfied. This often requires insertion of *nops* within and between VLIW instructions. In Section 2 we present the EVP and TriMedia VLIW architectures and describe the scheduling task for them in more detail. Section 3 presents how we adapted GCC to schedule correct VLIW code, presenting the use of the GCC's internal DFA-based scheduler for basic blocks, and two custom-made algorithms that we have developed: the *inter-basic block scheduling* pass guaranteeing that the scheduling constraints across the basic block boundaries are satisfied, and the resource-aware branch delay slot scheduler.

To achieve performance on a VLIW target, it is crucial that sufficient parallelism is exposed to the scheduling pass. In Section 4 we present several common VLIW compilation techniques used for this purpose and our GCC implementation of them. The techniques include loop unrolling precisely controlled by the unroll pragma and the address-based alias analysis. We remark that these techniques are applicable and can be beneficial for any processor that exploits ILP. In Section 5 we present the results which were achieved by implementing the aforementioned methods and report the performance of our GCC ports. Section 6 summarizes our conclusions.

2 Architectural Overview

The *instruction set architecture (ISA)* defines the compiler's (programmer's) view of a processor and its resources. The level of detail with which a processor is represented in an ISA can vary considerably. For the ISAs commonly used in general-purpose processors, this level is rather low, describing the registers, the memory and its addressing modes, and the instructions. The processor pipeline details are hidden from the compiler, which facilitates binary compatibility of different processors implementing the same ISA. When, for example, the pipeline depth, the instruction latencies, or the number of instructions that can be issued in parallel change in a new generation of a processor, old binaries can be executed without recompilation. This motivates the use of such ISAs in GPPs. We remark, however, that in this case the processor is responsible for the correct execution of the code. To achieve this, the processor contains a number of *interlocks*, which control the flow of instructions and data through the processor pipeline. An interlock is a piece of hardware which detects if an instruction at given pipeline stage can proceed to the next stage without causing *data hazards* and *structural hazards* [10] (also referred as *resource conflicts*). If this is not the case, the interlock hardware stalls (a part of) the processor pipeline till all hazards are resolved.

For VLIW processors in the embedded domain, binary compatibility is less of an issue, while the hardware complexity should be minimized. Therefore, a VLIW ISA employed there usually exposes more details of the processor pipeline organization to the compiler. Typically, the number of operations which can be encoded in a single VLIW instruction (referred as the number of *issue slots*) and the latencies of the operations are visible to the compiler. The responsibility of its scheduling pass is to bundle together the operations that can be executed in parallel without causing data or pipeline haz-

ards. The unoccupied issues slots in a VLIW instruction are padded with *nops*. The scheduler attempts to place the data-dependent operations in different instructions at sufficient distance from each other to avoid the data hazards. Similarly, it tries to schedule at sufficient distance the operations that occupy the same pipeline resource in order to avoid resource hazards. A VLIW machine often lacks (most of the) interlocks. The effect of a hardware interlock on the flow of instructions through the pipeline is similar to an insertion of a `nop` instruction in the program code. In case interlocks are absent, nop instructions should be explicitly inserted in the code by the compiler or the programmer. For a VLIW machine with a non-interlocked pipeline, the scheduling pass is responsible for inserting the nops between the VLIW instructions, so that all the hazards are avoided.

EVP is an 11-slot VLIW with a 9-stage non-interlocked pipeline. A single EVP VLIW instruction can issue in parallel up to 5 scalar operations and up to 6 vector operations. The individual operations can be predicated. We remark that in order to avoid excessive number of ports on the general-purpose register file (RF), EVP contains separate *pointer* (`ptr`) and *offset* (`ofs`) register files for address computations, and the *predicate* RF. Vector operations typically work on 16×256 bit general purpose vector registers from the `vr` RF. Due to register file port considerations mentioned above, the vector registers meant for specific type of vectors are contained in separate vector RFs, such as *vector shuffle pattern* RF `vsp`, *vector mask* RF `vm`, and several others. EVP is targeted at baseband signal processing, and one of its salient characteristics is the combination of the VLIW and vector processing with DSP features such as zero-overhead loops and circular addressing modes. For further details on EVP architecture, the reader is referred to [1].

TriMedia is a classic VLIW architecture which has been successfully implemented in several application processors targeted at multimedia domain. Its baseline instruction set consists of RISC-like operations working on a large register file consisting of 128×32 bit general-purpose registers (GPRs). These registers are used for arithmetic and memory operations, as well as for address calculations and predication. In addition to the typical RISC operations, TriMedia contains a rich set of special-purpose operations for media processing (referred as *customops*). Typically, they operate on GPRs seeing them as vector registers containing short vectors consisting of four 8-bit or two 16-bit elements. Recent generations provide customops operating on two concatenated GPRs, thereby increasing the vector length. Most of TriMedia processors are non-interlocked 5-slot VLIW [4].

3 Scheduling for an Exposed Pipeline VLIW Using GCC

GCC is primarily developed for processors with interlocked pipelines. Therefore, by default, GCC produces sequential assembly code and does not perform packing of VLIW instructions and insertion of nops, except for the nops in the branch delay slots. One way to provide a GCC-based compiler for an exposed VLIW processor would be by reusing the scheduler of an existing compiler for the machine and passing GCC's output through it. We have taken this approach for TriMedia, where GCC produces sequential code[1] for TriMedia ISA, which is scheduled afterwards by the separate *tmsched* scheduler [14].

Such an approach for EVP was not feasible because its scheduler was integrated in the proprietary CoSy-based compilation toolchain [5] and not available standalone. However, we were able to schedule correct and efficient VLIW code for EVP using the GCC framework by employing the internal GCC's scheduler and providing some additional functionality, as presented in the remaining part of this section.

3.1 DFA-Based VLIW Scheduling of Basic Blocks

Interlocked processors do not require instruction scheduling. Appropriate scheduling, however, facilitates reduction in the number of runtime interlocks and thereby improves performance. For this reason, GCC includes an instruction scheduler, which is implemented in the *haifa-sched* pass. The main algorithm performs top-down priority-based list scheduling on a *basic block (BB)* of RTL instructions and is implemented in the `schedule_block` function. At each cycle the algorithm attempts to schedule instructions which are *ready*, i.e., the instructions of which the data dependencies have been satisfied. In case a nonzero number of instructions have been scheduled at the current cycle, the mode of the first one is set to `TImode`. If no instruction have

[1] This, essentially, is the code for a 1-slot interlocked machine with TriMedia ISA.

been scheduled, we insert in the code a new RTL instruction (`const_int 0`) which represents a nop, and tag it with the `TImode` too. The nop insertion is done by the target hook `TARGET_SCHED_REORDER2`, which is called after an instruction is issued. It compares the current cycle with the cycle at which the previous instruction has been scheduled. If the cycles are not consecutive, it inserts an appropriate number of nops. Since the mode of all other instructions is `VOIDmode`, `TImode` tagged instructions represent the borders of VLIW instructions. In the final stage of the compiler we scan the instruction sequence using the `TImode` tags to emit assembly delimiters representing VLIW packing.

When the last predecessor of an instruction A is scheduled, the algorithm guarantees that no data hazards will happen by sufficiently delaying the cycle when A will become ready. The resource hazards are avoided when the algorithm attempts to schedule a ready instruction. It queries the *Deterministic Finite Automaton (DFA) based pipeline hazard recognizer* [9], which determines if scheduling an instruction at the current cycle will cause a resource conflict. In such a case the instruction is queued for the number of cycles needed to resolve the conflict. Pipeline resource descriptions used by DFA can be also utilized to specify other scheduling constraints. For example, the EVP instruction format provides only a single opcode field for a 32-bit *long* immediate. Therefore, only a single operation in a VLIW instruction can have such an a immediate operand, while several *short immediate* operands fitting in a narrower range are allowed. To satisfy this restriction, we specify a special pipeline resource representing the long immediate field in the opcode, and reserve it when scheduling an operation which requires such an immediate.

For EVP, due to a large number of issue slots and different instruction classes the generation time of a single DFA scheduling automaton became impractically large. In order to cut down the build time of the compiler the automaton was split into two separate automata: all functional units belonging to the scalar and address calculation parts of the processor are represented in the automaton `scalar`, and all vector functional units – in the `vector` automaton. This split resulted in two much smaller automata[2]. The majority of the instructions reserve resources from a single automaton. The only exception are certain vector operations that reserve resources from the `vector` automaton, as well as the long immediate unit which belongs to the `scalar` automaton.

[2]The combined number of states and arcs in the two automata is roughly 37000 and 250000, respectively. For a unified automaton, these numbers would be at least a 1000 times larger.

3.2 Inter-Basic Block Scheduler

The scheduling algorithm described above is applied to all basic blocks in the program's *Control Flow Graph (CFG)*. For each block B, it guarantees that the generated schedule will satisfy the constraints imposed by the operations belonging to B. This, however, is not sufficient for correct code generation. Suppose basic block A is an immediate predecessor of B in the CFG. Consider an operation $op_i \in A$ which is scheduled, for example, in the last cycle of A. Let $op_j \in B$ be an operation which uses the result of op_i. To satisfy the true data dependency, op_j should be scheduled not earlier than at cycle $latency(op_i) - 1$ (cycle counts in a schedule starts from zero). When scheduling B, the algorithm implemented in `schedule_block` is not aware of cross-block dependencies and might assign op_j to an earlier cycle, thereby creating an incorrect schedule. To fix such mistakes, we have implemented an additional *inter-basic block* scheduler.

For each basic block B, this function iterates over all of its instructions and checks whether dependent instructions located in all of the predecessor BBs are scheduled at sufficient distance. To find data dependent instructions from two blocks A and B, we make use of the GCC data structures which keep the live-in/live-out registers. Once two dependent instructions $op_i \in A$ and $op_j \in B$ are detected, the algorithm calls the backend-specific function `insn_latency(`op_i`,` op_j`)` which returns the number of cycles that have to be executed to satisfy the dependency. In case the distance between the instructions is too small, we insert an appropriate number of extra nops at the top of B.

Similarly to dependencies, resource reservations related to the scheduling of an instruction in one basic block may impose additional scheduling constraints for instructions in the subsequent blocks. To account for such effects, we provided an additional DFA-based algorithm which guarantees that all the resource conflicts are avoided. A similar mechanism have been also employed in our custom-made branch delay slot scheduler which is described below. Both schedulers operate on RTL code which has been already scheduled in GCC

sched2 pass, and take place in the customized reorg pass executed shortly before the final assembly emit pass [3].

3.3 Branch Delay Slot Scheduling

The existing GCC Branch Delay Scheduler (BDS) takes into account only the program data-dependencies and ignores the resource constraints, assuming that they will be ensured by hardware interlocks. To produce correct VLIW code, we disabled the original BDS pass and, initially, filled all delay slots with nops. However, the number of delay slots in EVP is quite large (5 or 7) and the associated performance penalty was considerable (e.g. $\approx -20\%$ for EEMBC-telecom). This made evident that a BDS pass is crucial to achieve satisfactory performance. The existing GCC BDS is a sophisticated algorithm and making it aware of resource constraints would be a rather challenging task. Hence, we decided to implement a proprietary resource-aware BDS for EVP. The algorithm iteratively attempts to move the branch up within the block. It starts with the last instruction preceding the branch and attempts to move the branch above it. This is equivalent to moving the instruction into a delay slot below the branch. If the move was successful, the algorithm tries to move the branch one more cycle up, proceeding in this was as long as moving is possible and there are non-filled delay slots. Each of aforementioned moves should not violate existing dependencies and should not create resource conflicts. The first condition is satisfied automatically: suppose instruction y depends on instruction x and y is moved into branch delay slot. Essentially, it means that the branch is inserted between x and y, thereby increasing the distance between them by 1 and satisfying the dependency.

Avoiding resource hazards requires more care. Suppose op_i is scheduled at cycle c_1 and uses resource r at cycle m and op_j is scheduled at cycle c_2 and uses resource r at cycle n. Consequently, resource r is used by op_i at cycle c_1+m and by op_j at cycle c_2+n. The GCC instruction scheduler ensures that $c_1+m \neq c_2+n$. Suppose that op_j is moved into a delay slot. As a consequence resource r is used by op_j at cycle c_1+n+1. In such case our BDS has to ensure that $c_1+m \neq c_2+n+1$. We solve this by making use of the GCC instruction scheduler; we artificially impose that op_i uses the resource r at cycle $n-1$ as well. For example in case of EVP the div operation which was using the register file write port at cycle 10, while the alu operations use it at cycle 3. Therefore, we impose that div uses the write port also at cycle 9. For an arbitrary processor resource utilization such solution may become too costly. However, for EVP we had to employ this technique only for a limited number of instructions. According to our benchmarking the change in the resource utilization of those instructions didn't induce performance penalty.

[3] The delay slot scheduler can introduce new nops thereby creating resource conflicts, as we illustrated above. Therefore, in our reorg pass it precedes the the inter-block scheduler, which bears the responsibility for guaranteeing the correctness of the final schedule.

3.4 Scheduling semantically equivalent operations

To achieve higher performance, EVP allows scheduling of some operations on different functional units. This allows several such operations to be scheduled in parallel in a single VLIW instruction. For example, moving data between two general purpose registers can be issued on two different functional units c_salu_1 or c_slsu_s1. These moves have different assembly syntax, move and move_slsu, and can be issued in parallel. Implementing this functionality using DFA caused the complications described below.

GCC emits an assembly mnemonic for an RTL instruction based solely on the set of operand constraints which it satisfies (recorded in GCC's which_alternative variable) and is agnostic of the instruction's DFA resource reservations. The two move instructions mentioned above have the same semantics and, therefore, their RTL templates and the operand constraints are identical. Hence, GCC has no means of differentiating them and emitting different mnemonics. To resolve this issue we have taken the following approach.

For an RTL instruction which can be issued on different functional units (with different mnemonics), during the sched2 pass we use the DFA to determine the functional unit on which the instruction was scheduled, and append to it an additional RTL pattern of the form: (clobber (match_operand N "const_int")), where the value N of the clobber operand is used to encode the information about the selected unit. This information stays attached to the instruction rtx till the final stage of compilation, allowing in this way that the proper assembly mnemonic is emitted. For example, the 16-bit move instruction can be issued on SALU and SLSU unit, and assembly generation is done as follows:

```
(define_insn "movhi_salu_or_slsu"
 [(set (match_operand:HI 0 "reg_operand" "=d")
       (match_operand:HI 1 "reg_operand" " d"))
 (clobber (match_operand 2 "const_operand" "=i"))]
 ""
 "*
  {
    switch (INTVAL(operands[2]))
    {
      case 22: return \"move %0, %1 %)%#\";
      case 33: return \"move_slsu %0, %1 %)%#\";
      default: break;
    }
    fatal_insn(\"something went wrong...\", insn);
  }"
)
```

In order to query the DFA automaton about scheduling decisions and to record this information in the clobber rtx, we have implemented the `evp_automaton_query` function and tied it to the target hook `TARGET_SCHED_DFA_NEW_CYCLE`, which is called everytime a new instruction is about to be scheduled. The main parameter of the function is `ready` rtx which represents the current instruction considered for scheduling. We illustrate the functionality of `evp_automaton_query` using the aforementioned 16-bit register move instruction as an example. In this case `ready` is RTL instruction of the form `(set (reg:HI ri) (reg:HI rj))`. First, we create two temporary RTL instructions, `insn1` and `insn2` which belong to `c_salu_1` and `c_slsu_s1` classes, respectively. Then we attempt to schedule each of these instructions at the current DFA state, by calling the `internal_state_transition()` function. Suppose scheduling of `insn2` was successful. This fact is memorized by assigning the variable `key=33`. Afterwards we trick the compiler into thinking that `ready` should be scheduled according to the `c_slsu_s1` pattern as shown below:

```
int uid = INSN_UID(ready);
dfa_insn_codes[uid] = internal_dfa_insn_code(insn2);
```

The DFA scheduler uses the array `dfa_insn_codes[]` (indexed by the instruction number `uid`) in order to store for each instruction its instruction class (and, hence, its resource reservations). The array elements represent the internal DFA codes of instruction classes which can be obtained by calling `internal_dfa_insn_code()`. By assigning the `dfa_insn_codes[]` for `ready` as shown in the code fragment above, we essentially force the compiler to think that the resource reservations of `ready` are as of the `c_slsu_s1` class, and to schedule it correspondingly.

The final action performed by `evp_automaton_query` just before exit is the attachment of `(clobber (const_int 33))` rtx to the original RTL pattern of `ready`. Upon exit from the function, the DFA scheduler will attempt to schedule `ready`. It will look up its instruction class in `dfa_insn_codes`, finding that it is `c_slsu_s1`, and will successfully schedule it.

We remark that in a case where all the alternatives contain resources from a single automaton, a different implementation of `evp_automaton_query` would be possible, based on the existing DFA facility which allows an instruction class to specify several scheduling alternatives using the OR construct (e.g. `"c_salu_1 | c_slsu_s1"`). Such an implementation, however, would lead to creation of a considerably larger DFA then in our method. Furthermore, in case the original DFA has been split into two or more automata (e.g., `scalar` and `vector` DFAs in case of EVP) and an instruction contains scheduling alternatives which belong to different automata, the DFA scheduler would not treat it correctly[4]. Our implementation, however, can treat such scheduling constraints properly.

4 Increasing ILP Exposed to the Scheduler

Majority of existing high-performance CPUs supported by GCC are superscalar processors, in which hardware mechanisms are employed to expose and exploit ILP. For example, branch prediction and speculation allow the processor's fetch and decode engines to run ahead of the execution and buffer decoded instructions from different basic blocks. In this way, the ILP across the basic block boundaries is exposed to the execution hardware which exploits it by issuing each cycle multiple instructions from the buffer, usually out-of-order, and guarantees that data dependencies and recourse constraints are respected. Effectively, it performs run-time scheduling. Hardware register renaming and dynamic memory disambiguation are often employed to remove false register and memory dependencies, thereby increasing the ILP and allowing more instructions to be issued in parallel.

For VLIW processors like EVP and TriMedia, the task of exposing and exploiting the ILP is shifted to the

[4]We have brought this issue to the attention of Vladimir Makarov, the developer and maintainer of DFA functionality. For the details we refer an interested reader to a corresponding discussion in the GCC mailing list.

scheduling pass of the compiler. Specific architecture features and compilation techniques are employed, allowing the scheduler to statically carry out the tasks which in superscalar processors are performed dynamically by the ILP hardware. For example, in order to remove the anti and output register dependencies, superscalar processors dynamically rename the compiler visible registers described in ISA to a larger set of hardware registers. To achieve similar effect in TriMedia, a much larger set of registers is provided directly in the ISA, and the compiler statically renames the registers that cause false dependencies.

The scheduler operates on a certain *scheduling scope*, also referred as *scheduling unit*. This scope is usually given in terms of basic blocks, and can range from a single BB to the complete CFG. As the ILP available in a single block is limited, superscalar processors employ branch prediction and speculation to discover the ILP across the block boundaries. To achieve a similar effect, scheduling for VLIW machines is performed on multiblock scheduling units. For example, a TriMedia scheduling unit is a *decision tree (dtree)*, which is a CFG subgraph with the single entry and multiple exits [14][5].

The task of a VLIW scheduler is to assign to each operation *op* in the scheduling unit an integer $c(op) \geq 0$, which denotes the order of the VLIW bundle to which it belongs. This integer is also the number of the cycle (counted from the beginning of scheduling unit execution) at which the operation will be issued. The generated parallel code should preserve the semantics of original program. To achieve this, the scheduler detects data and control dependencies between the operations and schedules them such that the dependencies are preserved.

The quality of the final schedule depends on two main factors: the amount of ILP present in the scheduling unit and the capability of the scheduling algorithm to extract and utilize this parallelism. The latter factor constitutes a complex subject for a standalone study which falls outside the scope of this paper. Therefore, the remaining part of this section is dedicated to a number of techniques which increase the amount of ILP and describes how they were supported in our GCC ports for EVP and TriMedia. Two approaches are commonly used to expose more ILP:

- **Scheduling scope increase** provides the scheduler with larger number of operations, and therefore increases the chance to find the independent ones, which can be executed in parallel.
- **Reducing dependencies** between operations increases the scheduling freedom thereby increasing the chance to schedule them in parallel.

The enhancement to GCC *loop-unrolling* which allows scheduling scope increase in a precisely controlled fashion is presented in Section 4.1. The scheduling scope in our port has been also increased by application of if-conversion and tail duplication. For these passes, however, we employed the existing GCC implementations, which have certain limitations. GCC tail duplication pass, for example, applies the transformation relying on internal compiler heuristics and does not provide direct control to a programmer. Development of such functionality would enhance this technique and constitutes an interesting subject for the future work. Section 4.2 describes the *address-based alias analysis* on the RTL which we implemented in order to improve the existing GCC memory disambiguation capabilities. Alias analysis allows static disambiguation of memory accesses thereby reducing number of false dependencies between them.

4.1 Controlled Loop Unrolling

Loop unrolling is a common code transformation which replicates the loop body several times. It creates a larger segment of non-loop code and, consequently, facilitates creation of a larger scheduling scope. Additionally, it decreases the number of updates of induction variables and the number of loop exit tests. GCC contains two unrolling phases: the first one works at the Gimple level and does total loop unrolling while the second one operates at the RTL level and does partial unrolling. These phases did not completely suit our needs and had the following limitations. The total loop unroll phase requires the iteration count to be a statically known constant, which is not always the case. Furthermore, the code size penalty resulting from the total unroll can be unacceptable. Partial unrolling is more suitable for our purposes. However, the corresponding GCC phase induces the unroll factor based on heuristics, and can be controlled by the programmer only indirectly by means of the following hooks: `PARAM_MAX_AVERAGE_`

[5] We remark that GCC supports *Extended Basic Block (EBB)* scheduling units; an EBB is a rudimentary form of a superblock and hence, different than a dtree.

`UNROLLED_INSNS`, `PARAM_MAX_UNROLL_TIMES`, and `PARAM_MAX_UNROLL_TIMES`. Despite using these facilities, we were not able to steer the compiler towards achieving the optimal unroll factor for all the cases and, consequently, observed significant performance penalties on certain benchmarks. A TriMedia or EVP programmer chooses the unroll factor very carefully. Insufficient unrolling does not expose enough parallelism. Excessive unrolling, on the other hand, leads to increased code size and creates too much register pressure, which results in spills and performance degradation. In TrimMedia, the selected unroll factor is communicated to the compiler by means of the unroll pragma of the following form: `#pragma TCS_unroll n`, where `n` represents the unroll factor. This pragma is heavily utilized for optimization of the production code.

As the support for such precisely controlled unrolling was missing from GCC, we have added it in our backend. Initially, we have considered the GCC facility for adding attributes which could have potentially being extended to support the unroll pragma. However, currently the attributes can be only attached to functions and not to the loops. Therefore, a different approach has been taken, as described below. Instead of an attribute, we attach to a loop a new special-purpose *unroll* RTL instruction which holds the unroll factor. During the RTL unroll phase, we analyze each loop and, when present, retrieve the associated unroll instruction. The unroll factor is extracted and applied to the loop, and the instruction is discarded.

The association between the pragma and the special RTL instruction is realized as follows. First, we add a new built-in function `__unroll_pragma()`, which has a single integer parameter representing the unroll factor. Second, the `REGISTER_TARGET_PRAGMAS` hook is employed to introduce the new unroll pragma to the compiler. The `trimedia_unroll_pragma()` function is tied to this hook and is called during parsing each time when the pragma is encountered in the source code. This function substitutes the pragma with a call to `__unroll_pragma()`. Finally, during the RTL expansion, the call to the builtin is substituted with the unroll instruction RTL:

```
(define_insn "customop_unroll_pragma"
 [(unspec_volatile:SI
      [(match_operand:SI 0 "immediate_operand" "i")
      ] UNSPEC_unroll_pragma)
 ]
 ""
 ""
)
```

We remark that the instruction is declared as `unspec_volatile` in order to avoid it being moved away from the corresponding loop during the optimization passes.

4.2 Address-Based Alias Analysis on RTL

Alias analysis (AA) is a technique that allows to recognize if two pointers do not refer to the same address (i.e., alias). Stronger alias analysis allows to reduce the number of dependencies between memory operations in a scheduling unit. This increases amount of ILP that can be utilized and, potentially, leads to a shorter schedule. Strong AA is particularly important for making loop unrolling and software pipelining to be effective on a VLIW machine. In this techniques, scheduling scope consists of operations belonging to several loop iterations. Consequently, memory operations from different iterations will be present in the scope. If AA is weak, spurious dependencies will be created between the memory operations. These dependencies limit the scheduling freedom and the amount of cross-iteration ILP that can be utilized.

GCC provides AA support at both the GIMPLE and the RTL level. The Gimple AA has been introduced within the Tree SSA infrastructure, while the RTL AA is due to the old (before version 4.0) RTL-based infrastructure. The Gimple AA includes *type-based analysis* and *points-to analysis*. Type-based analysis makes use of the C language aliasing rules. It checks the pointer types of two memory accesses and, in case they are different, concludes that the accesses are disjoint[6].

Points-to analysis (or pointer analysis), is a technique that establishes to which variables or storage locations an arbitrary pointer points to. The variables or storage locations are united into sets, which afterwards are used to disambiguate arbitrary pointers. Using the code fragment below we illustrate the capabilities of the points-to analysis.

```
int *p, *t;
int a[10], b[10], c[10];

 if (d > 10)
     p = b;
```

[6] Exceptions: 1) one may use a pointer or reference to a signed type to access an object of unsigned type, or vice versa, 2) one can use a pointer or reference with different const-ness or volatile-ness than the object, and 3) one can use a pointer of type char or unsigned char to access any object.

```
else
    p = c;
t = a;
for (int i = 0; i < d; i++)
    *(t+i) = *(p+i);
```

Based on this analysis GCC correctly finds out that `p` points to the set $\{b,c\}$ and `t` to the set $\{a\}$. As the sets are disjoint, GCC concludes that the two pointers do not alias. We remark that when points-to analysis is employed for an unrolled loop, its capabilities are limited. It will be able to disambiguate unrolled stores (i.e., `*(t+0),*(t+1)...`) from unrolled loads (i.e., `*(p+0),*(p+1)...`). However, it will not be able to disambiguate among different stores. As points-to analysis is performed at GIMPLE level, it can only be employed for the loop totally unrolled using the GCC unroll pass on the GIMPLE level.

In Section 4.1 we have presented why partial and precisely controlled loop unrolling on the RTL level is desirable for a VLIW machine. After implementing this functionality in our port, we have observed that the performance gains were limited. The reason for this is the weakness of existing AA on the RTL level. The current RTL AA is mostly type-based and therefore, can not disambiguate the stores `*(t+0),*(t+1)...` from the loads `*(p+0),*(p+1)...`. Consequently, although the loop gets unrolled, little or no cross iteration ILP is extracted. The purpose of the work presented in this section is to improve AA on RTL, so that the benefits of RTL loop unrolling for VLIW scheduling can be reaped [7].

Improving the RTL AA can be done in many ways. One option consists of improving the transfer of information between Gimple and RTL. To achieve this, one has to adapt the alias information model used by the two compiler representations: Gimple uses explicit representation in terms of points-to sets, while RTL is relies on a query-based disambiguation, i.e., whenever two memory references are to be disambiguated, an alias problem is formulated and solved. Propagation of AA information from Gimple to RTL has been addressed in [12] and implemented by a GCC patch and by a separate GCC branch. We have tried both implementations but were not able to obtain expected execution performance; in fact we observed a small performance decrease. This could have been caused by the following reason. In order to be effective for the case of partial loop unrolling on the RTL level, next to propagating the alias information from a GIMPLE representation (where the loop has not yet been unrolled), the algorithm would have to additionally disambiguate each newly introduced RTL memory statement. Such functionality was missing in the patch.

An alternative to propagating the alias information from Gimple to RTL is to enhance the AA on the RTL level. To achieve this we added flow-sensitive *address-based alias analysis* at the RTL level. Prior to our work, a similar approach has been proposed in [13]. The corresponding patch, however, has never been added to the GCC mainline due to associated increase in compilation time. Furthermore, this approach has the following drawbacks:

1. The technique is based on the idea of representing a memory address by means of an *address descriptor*, which is a pair <I,Z>, where I is an operation and Z is a mod-k residue set. An address descriptor can keep track of only one operation (i.e., I). Due to this limitation, this approach is not able to disambiguate addresses obtained by linear combinations of values generated by more then one operation.

2. The technique extracts alias information across loop iterations, which leads to a significant increase in the compilation time. However, as pointed out in [11], the GCC internal scheduler deals with acyclic graph regions and, therefore, the extraction of alias information across loop iteration is of no use.

Our flow-sensitive address-based alias analysis overcomes the aforementioned limitations allowing to disambiguated memory accesses present within the same basic block. The analysis can be sketched as follows: Suppose a memory access part of an operation op which belongs to a basic block BB_s. The address of the memory access, is given by an original linear function f. By starting from op and traversing in revers order the BB_s operations, f is composed with the linear expressions representing dependent operations. In the end we obtain a final linear function which represents the address in terms of regs defined outside the BB. Afterwords, the composition is continued over a number of control

[7] Other RTL passes such as CSE, DSE, GCSE, and register allocation make use of alias information calculated at RTL level and would also benefit from more powerful AA.

paths; for each such path i a corresponding linear function f_i being derived. Those control paths are obtained as follows:

1. **non-backedge paths:**

   Starting from BB_s compose in reverse order over single predecessor blocks. Additionally we impose the limitation that among those single predecessor blocks at most one of them sources/sinks backedges. Once a BB with more more than one predecessor is encountered, duplicate f, one for each of the predecessors and continue for as long as single predecessor basic blocks are encountered none of those blocks sourcing/sinking back-edges.

2. **backedge paths:**

   Starting from BB_s compose in reverse order over single predecessor blocks. Once a BB with more than one predecessor over incoming backedges is encountered duplicate f, one for each of the predecessors and continue for as long as single predecessor without incoming/outgoing back-edges blocks are encountered. Once a BB with more more than one predecessor is encountered duplicate f, one for each of the predecessors and continue for as long as single predecessor basic blocks are encountered, none of them sourcing/sinking back-edges nor the original BB_s.

Once the linear functions f_i are derived for every memory access op, two different accesses op_m and op_n from the same BB can be disambiguated. The two operations do not alias if: $\forall i, f_i(op_m) - f(op_n) \neq 0$.

Example Consider in Figure 1 a memory operation that belongs to the basic block E. As a result of our CFG traversal 5 linear functions corresponding to 2 non-backedge and 3 backedge paths will be generated. Those paths are as follows:

the **non-backedge paths** consist of the following basic blocks: $\{E,D,C,A,A1\}$ and $\{E,D,C,B,B1\}$.

the **backedge paths** consist of the following basic blocks: $\{E,D,I,F\}$, $\{E,D,I,G\}$ and $\{E,D,H\}$.

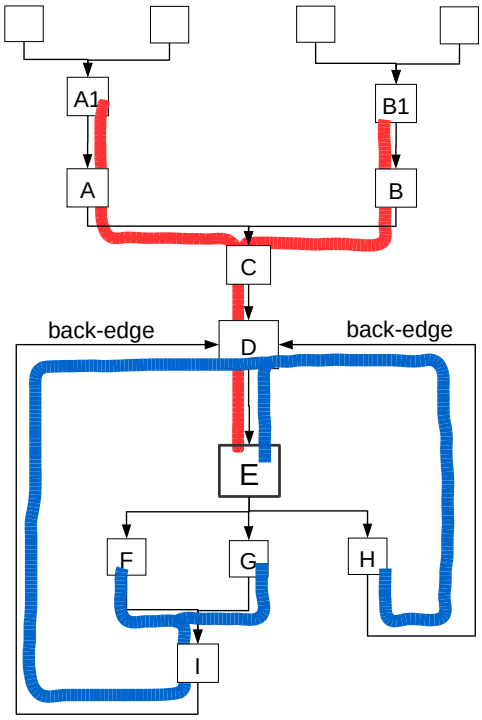

Figure 1: CFG traversal during alias analysis: with red/blue you can see the non-backedge/backedge paths.

5 Experimental Results

The GCC ports for EVP and TriMedia have been compared to the existing compilers supporting these processors. For TriMedia, our *tmGCC* port has been compared to the production *tmcc* compiler which is a part of the *TriMedia Compilation System (TCS)* [6]. In the TCS toolchain, the core compiler *tmcc* generates the sequential code, and splits it into scheduling units, called *decision trees (dtrees)*. The output of *tmcc* is then passed to a standalone VLIW scheduler *tmsched* [14]. According to the TCS convention, *tmcc* performs register allocation only for *global registers* which constitute a half of the complete register file. The *tmsched* scheduler performs register allocation of the remaining 64 *local* registers, peephole optimizations, and VLIW scheduling. Scheduled code then goes through the standard assembling and linking procedure to obtain the binary, which is then simulated or run on the target hardware to obtain the performance data. In order to generate scheduled TriMedia code with GCC, we follow a similar approach. Our *tmGCC* port acts as a core compiler generating sequential assembly, which is then formatted according to *tmsched* requirements, and passed to it for schedul-

ing. After scheduling, the final binary is generated in the same way as for *tmcc*.

We have experimented with a set of representative benchmarks from the media/signal processing domain. Our *mediastone* testsuite contains audiocodecs (*ac3, mp3,dts*), video algorithms (*MPEG2*, motion compensation), image processing kernels (color conversions and filters), and the complete *EEMBC telecom* suite. The *perf-baseline* suite contains several proprietary algorithms for image and video improvement, such as *up-conversion*, and custom implementation of MPEG-2 and h264 video codecs. We remark that these are product quality applications, where a significant part of the code was hand optimized by programmers.

In an embedded system, the most important factors for compiler evaluation are the performance and code size of the generated code, while the compilation time is not critical. Table 1 presents these metrics for the code produced using both GCC and the production compiler, *tmcc*. For brevity, we report the results only for the *tm3271* core. The results for the *tm3260*, *tm5250* and *tm3282* were similar. In the table, *tmGCC-old* refers to the prototype TriMedia port developed at Philips Research earlier this decade. This prototype was not mature enough, lacking the ILP-enhancement features described in Section 4. It was relying completely on *tmsched* for ILP extraction and, essentially, can be seen as a port to a single-issue processor with TriMedia ISA. As depicted in the table, compared to *tmcc*, *tmGCC-old* exhibits severe performance degradations, 23.7% and 39.7%, showing that a bare GCC port without specific VLIW support is not well-suited for TriMedia, even when it is coupled to a mature VLIW scheduler. We remark that in addition to ILP-enhancement techniques, *tmGCC-old* is missing support for some of the addressing modes and for the new custom vector operations. This explains higher grade of degradation on *perf-baseline*, as a large portion of its applications was manually rewritten to utilize the new operations.

The numbers in the third column of the table represent the results obtained with *tmGCC-current*, the current port of GCC for TriMedia. We remark that in addition to the presented techniques, *tmGCC-current* contains support for vector operations missing in *tmGCC-old* and some enhancements to the GCC software pipelining and if-conversion passes. These techniques are currently under development by the TCS compiler team and are not reported in this paper. The presented results show that

Testsuite	tmGCC-old	tmGCC-current	tmcc
mediastone			
cycles	123.7%	102.8%	100.0%
code size	100.3%	110.2%	100.0%
perf-baseline			
cycles	139.7%	105.9%	100.0%
code size	99.9%	102.5%	100.0%

Table 1: Relative Performance of the GCC ports and the production compiler for TriMedia.

implementing the ILP-enhancing features exposes more ILP to *tmsched*, allowing it to dramatically improve the performance of the scheduled code. The performance gap with the mature production compiler is reduced to just 2.8% on *mediastone* and to 5.9% on *perf-baseline*. However, the performance improvement for mediastone is achieved at a cost of a 10% code size increase. We suppose that the loop-unrolling in our applications could have been too aggressive. The unroll factors in the benchmarks were chosen by the programmers to provide best performance with *tmcc*. These factors could be not optimal when compiling with *tmGCC*. Carefully selecting these factors would, probably, resolve this issue, but such work fell out of the scope of our study project. We have also identified another source of potential improvement for the GCC port. Over the years of development, a large number of peephole optimizations has been introduced to *tmcc*. Our GCC port lacks the vast majority of these peepholes. We remark that the GCC facilities for peephole optimizations have limitations which do not allow all the *tmcc* peepholes to be easily introduced. Namely, the `peephole2` pass of GCC handles only adjacent operations. The operations which are non-adjacent, but connected by a data dependency can be handled by the *combine* pass. This pass, however, considers for optimizations only triples of operations connected by data dependencies, such that the dependence graph is linear. Allowing more generic forms of graphs would be desirable and could increase the power of peephole optimization pass of GCC.

Similarly to experiments reported above, we have compared our GCC port for EVP with the current EVP production compiler. We remark that, differently from TriMedia case, our EVP port performs also VLIW scheduling. To achieve correct and efficient VLIW code generation, we have implemented the techniques presented in Section 3. The comparisons were performed on the standard benchmark for telecommunication industry,

EEMBC-telecom [8] and encouraging results have been obtained. Concerning our EVP GCC port (*evpGCC*), we would like to report an interesting experiment related to a comparative performance of compiler-optimized and hand-optimized code. In embedded systems the quality of the application code directly affects the cost and performance of the final product, thereby motivating significant amount of optimization effort done by programmers. This is particularly true for the telecommunication domain where programming using assembly or compiler intrinsics is still common. Although such programming model is costly, the numbers presented below provide a reason for this approach and motivation for improving the power of compiler technology. For our experiment we consider a 256-point complex FFT algorithm. First, a standard C implementation from EEMBC-telecom is taken, and is compiled using *evpGCC*. Second, we compile the tailor-made version of the algorithm version manually optimized for EVP [7]. Both implementations are simulated using the EVP simulator. The obtained results are depicted in Table2, which shows that manual optimization provide a performance improvement by a factor of 122×.

application	standard FFT	optimized FFT
FFT		
cycles	121086	986
code size (bytes)	3512	4936

Table 2: Performance and Code size for 256-point complex FFT.

A factor of 16 out of 122 can be explained by the inability of GCC's vectorizer to vectorize the FFT code. In particular, the vector shuffle patterns employed in optimized implementation are hard to be auto-generated by the compiler. In fact, a factor significantly larger than 16 can be attributed to the absence of vectorization for the following reason. In the optimized version, the FFT butterfly data rearrangements are performed on the vector registers, whereas for the non-optimized code they are done via memory. This requires excessive memory traffic making the load/store unit a bottleneck. The dramatic performance gap suggests that, apart of vectorization, a number of other compiler techniques were not effective. We were able to identify several such cases. First, we observed that on the standard code, loop unrolling has not been performed by GCC, while in the optimized version the already vectorized loop body was further unrolled manually 4 times. The presented example, in our opinion, provides a challenging testcase for compiler engineers and may allow them to identify the weaknesses of existing optimizations.

6 Conclusions

The goal of the GCC porting projects presented in this paper was to evaluate suitability of GCC for code generation for non-interlocked VLIW processors. Out conclusions can be summarized as follows.

First, the obtained results illustrate that GCC can be used in a VLIW compilation toolchain, both as a core compiler coupled to an external VLIW scheduler, and as the complete solution performing both sequential code generation and VLIW scheduling. For TriMedia, which represents the former case, where our GCC port could benefit from a mature VLIW scheduler, the results were particularly encouraging, and the decision to productize our *tmGCC* prototype has been taken. Second, we have identified several areas where current GCC can be strengthened to better support VLIW compilation. Namely, GCC loop unrolling and alias analysis on the RTL can be improved to increase the amount of exposed ILP. Furthermore, DFA mechanisms in GCC have limitations when handling processor with significant number of instructions with several scheduling alternatives. Finally, GCC facilities for peephole optimizations, `peephole2_optimize` and `combine` have limitations, alleviating which could improve performance of both VLIW and non-VLIW targets.

In our ports we have developed partial solutions to some of the identified issues. However, development and integration of general solutions in the GCC framework will be of interest for the compiler engineers in the embedded domain considering to use GCC as a compiler framework for their VLIW (or non-interlocked pipelined) targets. In particular, our approach for inter basic block scheduling and for resource-aware delay slot scheduling can be improved.

In our opinion, some of the solutions implemented in the EVP and TriMedia ports could be beneficial to a wider range of GCC targets. First, the techniques which increase the amount of exposed ILP, such as the controlled loop unrolling and the addresses-based alias analysis on the RTL level, could be beneficial for non-VLIW processors that exploit ILP, e.g., superscalar or deeply pipelined scalar processors. Second, the DFA-related techniques presented in Section 3.4 allow dramatic reduction in the DFA size and generation time for the

cases when significant number of target instructions have multiple scheduling options and explosive growth of DFA is observed. Furthermore, in case the original DFA automaton for a processor has been factorized, our approach allows correct scheduling of instructions which contain alternatives from two different DFA automata.

Acknowledgments. We would like to thank our former colleagues from TriMedia TCS Compiler Team at NXP Semiconductors for the fruitful collaboration during the *tmGCC* study project and for generously providing us the experimental results for the current *tmGCC* port. We thank Jan Hoogerbrugge, Norbert Esser and Willem Mallon for their work on the initial *tmGCC* port. We also thank Wim Kloosterhuis for the discussions on EVP and for providing the production-quality FFT implementation optimized for EVP. Finally, we would like to thank Vladimir Makarov and other people in the GCC community for sharing their knowledge with us on the GCC mailing list.

References

[1] Philips Semiconductors Adelante VD32040 architecture: An embedded vector processor for low power DSP applications. Jean-Paul Smeets, Kees Moerman, *Proc. of GSPx 2005*.

[2] Kees van Berkel, Frank Heinle, Patrick P. E.Meuwissen, Kees Moerman, and Matthias Weiss. Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices. *EURASIP Journal on Applied Signal Processing 2005:16, 2613.2625*.

[3] G. A. Slavenburg, S. Rathnam, and H. Dijkstra. The TriMedia TM-1 PCI VLIW Mediaprocessor. *Hot Chips 8*, 1996.

[4] J.W. van de Waerdt, The TM3270 Mediaprocessor, PhD Thesis, Delft University of Technology, 2006.

[5] The CoSy compiler development system, `http://www.ace.nl`

[6] TCS The Trimedia compiler system, `http://www.tcshelp.com`

[7] Mahima Smriti, Kees Moerman. A Generic Self-sorting FFT Implementation on the VD32040 Vector Processor. *Proc. of GSPx, 2006*.

[8] EEMBC, `http://www.eembc.com`

[9] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. *Proc. of GCC Summit, 2003*.

[10] John Hennessy and David Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA,

[11] Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for IA-64 *Proc. of GCC Developer's Summit*, pp. 1-13, Ottawa, Canada, 2005.

[12] Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin. An approach for data propagation from Tree SSA to RTL, *Proc. of Int. Workshop on GCC for Research in Embedded and Parallel Systems (GREPS)*, Brasov, Romania, 2007

[13] Sanjiv K. Gupta and Naveen Sharma. Alias Analysis for Intermediate Code, *Proceedings of the GCC Developer's Summit*, 2003, pages 7178.

[14] Jan Hoogerbrugge and Lex Augusteijn, Instruction scheduling for TriMedia, *Journal of Instruction-Level Parallelism*, 1999, vol. 1.