Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

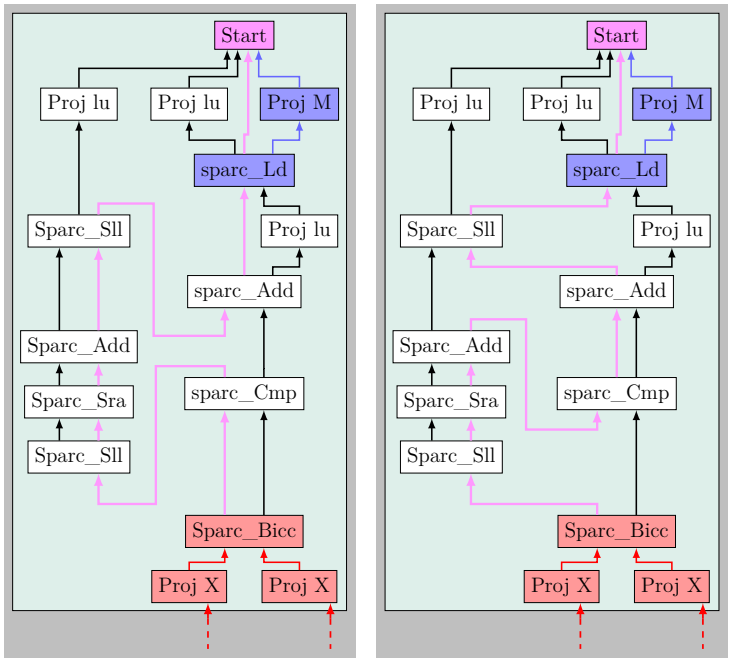
Lehrstuhl Prof. Dr.-Ing. Snelting

Pipeline-aware Scheduling for the LEON3 Processor in libFirm

Bachelorarbeit von

Philipp Schaback

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Andreas-Fried
Abgabedatum:	24. März 2020

Abstract

Die meisten modernen Prozessoren sind um eine Instruktionspipeline herum gebaut. Moderne Compiler produzieren aber nicht immer Maschinencode, welcher diese Pipeline maximal ausnutzt. Die resultierenden Kompilate verschwenden Prozessorzyklen. In diesem Paper betrachten wir einen alternativen Schedulingansatz, welcher auf eine maximale Pipelineauslastung optimiert ist. Anschließend bewerten wir dessen Resultate auf einem SPARC V8 - LEON3 Prozessor. Die erreichten Verbesserungen betragen bis zu ~3% in manchen Tests.

Most modern processors are built around an instruction pipeline at their core. Modern compilers however don't schedule instructions utilizing this pipeline to its fullest potential. The resulting pipeline errors waste processor cycles unnecessarily. This paper explores an alternative scheduling approach which focuses on reducing these errors and evaluates its performance on the SPARC V8 - LEON3 Processor, finding improvements of up to ~3% in some tests.

Contents

1 Introduction

Most modern processors are build with pipelines as an underlying microarchitecture [1]. The execution of an instruction is broken down in multiple stages which are stepped through sequentially. Once an instruction has completed one such stage, it is moved to the next and the following instruction may proceed to the previously occupied stage. This allows for multiple instructions to be executed in parallel, albeit shifted by one stage. However, problems arise if a later instruction depends on the result of a preceding one: Waiting for its completion negates the advantages of the pipeline, effectively reverting back to a purely sequential model. Architecture designers combat this by introducing backtracking paths from later stages to previous ones, allowing for less delay waiting for results of previous instructions. For example once a calculation is done, its result may be available for the proceeding instruction immediately, without writing it back to a register and later reading it from there again. Some instruction pairings however still incur additional delay. By ordering instructions with the timings of the instruction pipeline in mind this delay can be reduced or mitigated. In this paper we propose an alternative scheduler doing just this for the libFirm compiler, by interleaving independent instructions into such pairings, breaking up the delay-producing dependencies. We explore this concept on the LEON3 Processor, an open source processor design made available for free by Cobham Gaisler under the GNU GPL license [2].

First, we explain the prerequisites necessary for this thesis in chapter 2. Then we introduce the problem at hand, our theoretical solution to it and present our practical implementation in chapter 3. After that, we evaluate our solution using static analysis and dynamic measurements in chapter 4. Finally, we close with a critical view of our results and propose further optimizations possible in chapter 5.

2 Basics

First, we are going to introduce some basic concepts necessary for this paper:

2.1 Compilers

A compiler's job is the translation of human-readable and human-writable, high-level code into instructions executable by a processor. Like most large software projects, compilers are built with a layered architecture increasing maintainability and reducing code duplication amongst other advantages. Sharing data between layers is handled through a a common language, the *intermediate representation (IR)*. Generally a compiler following this approach is constructed from three layers: The *front-ends* translate from a higher level language into the IR, the *middle-end* applies generic optimizations and the *back-end* applies further target-specific optimizations and emits binaries runnable on the target. Each of these layers consists of multiple stages run through in sequence. For the back-end, theses are the most important ones: The *code selection* phase searches for known patterns in the IR, replacing them with target specific instructions. The *scheduling* phase works out the execution path through the instructions. It will be discussed in more detail in the following section. In the *register allocation* phase generic registers used as operands are replaced by specific ones available in the target architecture. Finally, the *assembly emission* phase converts the IR into actual assembly runnable on the target.

libFirm is the compiler middle- and back end used and modified by this paper. It is combined with different front ends to form a fully-featured compiler, like the cparser front end for the C language.

2.1.1 Intermediate Representation (IR)

The intermediate representation is the common language used by all layers and phases of the compiler. The compiler front-end translates the source language into the IR. Any later compiler stage (including scheduling) operates on this IR only.

One property an IR can uphold is *static single assignment (SSA)*. With SSA every variable has to be assigned once and only once and must be defined before first use. If a variable is originally assigned to multiple times, converting it to SSA Form will result in multiple differently named copies as seen in Figure 2.1. The ϕ-Operator merges variables from different control flows by choosing one or the other based on the path taken. SSA Form allows for easier implementation of many optimization strategies as finding the definition and value of any variable becomes trivial.

```
a := 3
if a > 2
        b := 1
else
        b := 3
c := a + b
```

(a) Pre-SSA code

```
a1 := 3
if a1 > 2
        b1 := 1
else
        b2 := 3
b3 := φ(b1, b2)
c1 := a1 + b3
```

(b) Post-SSA code

Figure 2.1: Example code and its equivalent in SSA form

A *control flow graph (CFG)* describes all possible paths execution flow could take in a program. It consists of blocks and directed edges connecting those blocks. Each block is made up of a set of instructions that are to be executed in sequence, and must not contain any jumps, branches, nor jump targets. The edges model execution paths between blocks.

Firm is a graph-based intermediate representation [3] used by the libFirm compiler [4]. Its graph is a combination of a CFG, as well as a data dependency graph.

It upholds SSA at any point in time by design since dependencies are represented by edges, which can only point to one node at a time, thus not allowing multiple definitions for the same variable.

2.1.2 Scheduling

The compiler phase most important to this thesis is the scheduling phase, since it determines the ordering of instructions. In libFirm, scheduling is performed on a block-by-block basis. While optimizing for best resource utilization is obviously the goal, computing a perfect schedule is infeasible. Different schedulers approach this goal with different approaches: libFirm's default scheduler optimizes for minimal number of registers used. This is a sensible goal since writing and later reading register contents to and from memory (*spilling registers*) unnecessarily wastes time. Many programs however don't need all registers, thus never coming close to register spilling. In such cases, different scheduling algorithms, like the one proposed, can result in better utilization of resources.

2.2 Instruction Pipeline

The instruction pipeline is a central part of most modern processors [1]. The pipeline consists of multiple stages through which each instruction steps sequentially. Once an instruction has completed one such stage, it is moved to the next and the following instruction may proceed to the previously occupied stage. This allows for overall faster execution time since multiple instructions are executed in parallel albeit shifted by

one stage. The exact makeup of the pipeline varies from architecture to architecture. Most pipelines however include the following stages: In the *instruction fetch stage (IF)* the next instruction to be executed is read from memory and the instruction pointer is incremented. The *decode stage (DE)* decodes the instruction and fetches the required operands from the registers. The *execution stage (EX)* performs the actual calculation required by the instruction. Finally, in the *write-back stage (WB)* results are written back into registers. Depending on the architecture, reading and writing memory, emitting exceptions and other auxiliary tasks are either integrated directly into these stage or externalized into their own stage each. Problems arise however if one instructions operands depend on the result of a previous instruction, as seen in Figure 2.2:

```
ldi $rax, 3
add $rax, $rcx
```

Figure 2.2: Example code showing one instruction depending on the previous one: The *add* instruction uses the *$rax* register, which is written by the immediately preceding *ldi* instruction. This could produce a hazard, depending on the architecture of the pipeline.

When the add instruction reaches the decode stage, the load instruction is in the executing stage. The value of *$rax* has not been set to 3 yet as the load instruction has not reached write-back. The result of the computation would be wrong. To combat this the processor introduces a pipeline stall to give the load instruction time to write into *$rax*. It does so by inserting so-called *bubbles* after the load, delaying each following instruction by one stage. When a bubble reaches a stage, it is ignored processing-wise, introducing effectively only a delay. This process can be seen in Figure 2.3. The purple instruction depends on the result of the green instruction which is not ready when needed, so the processor inserts a bubble. The execution takes one cycle longer.

Since instructions depending on each other is extremely common, most processor include hardware remedies for some of these stalls. Backtracking paths from later stages to previous once allow later instructions to use results without waiting for write-back and then reading from the register themselves. Such paths would make the result of the load available as soon as possible, allowing the add to be executed without any pipeline stalls.

Some pipeline stalls however can't be remedied by the processor in this way and still remain. These can only be reduced by ordering the instructions differently.

[1]Derivative of `https://commons.wikimedia.org/wiki/File:Pipeline,_4_stage_with_bubble.svg` licensed under CC-BY-SA-3.0. Original by Colin M.L. Burnett licensed under CC-BY-SA-3.0.

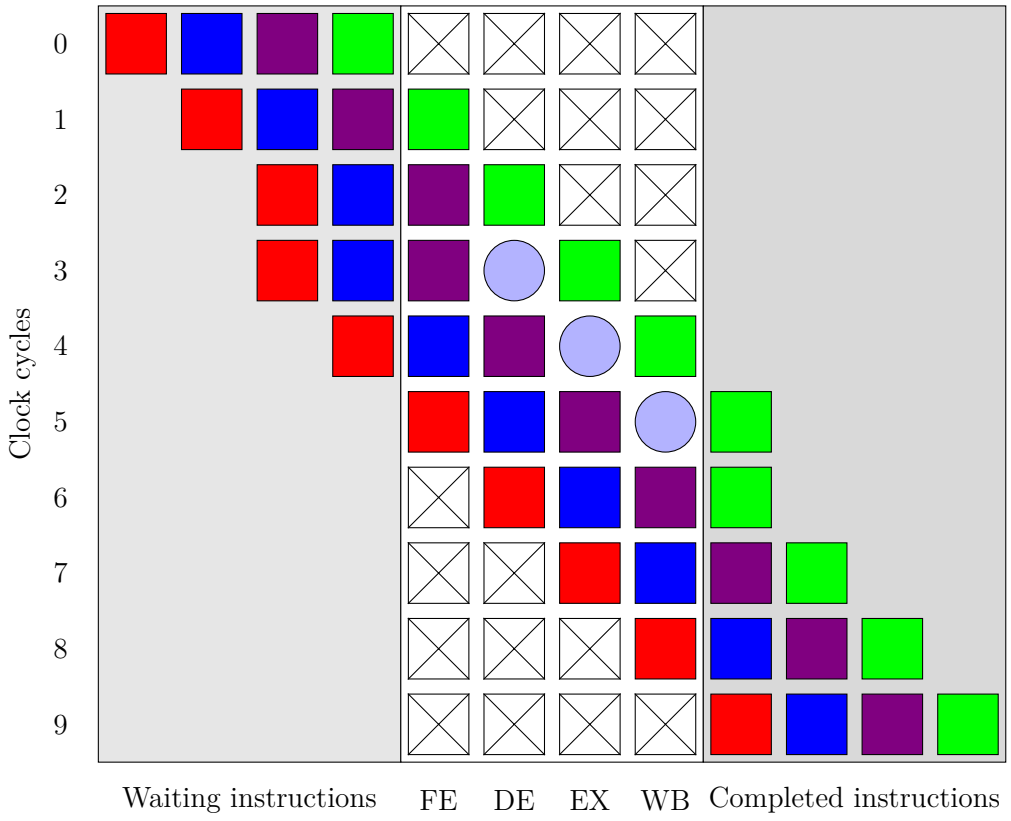


Figure 2.3: Simple five stage instruction pipeline exhibiting bubbling: The purple instruction depends on the result of the green instruction which is not ready when needed, so the processor inserts a bubble. The execution takes one cycle longer. [1]

2.3 SPARC Architecture & LEON3

The LEON3 is a 'synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture' [5]. Its instruction pipeline consists of seven stages: Instruction Fetch (FE), Decode (DE), Register Access (RA), Execution (EX), Memory (ME), Exception (XC) and Write (WR). It is also equipped with backtracking paths XC $\Rightarrow$ RA, XC $\Rightarrow$ EX, ME $\Rightarrow$ RA and EX $\Rightarrow$ RA [6, Section 83.2.1], which keep most dependency pairs hazard-free as explained in section 2.2. Which instruction pairing still produce hazards will be discussed in the next chapter. The LEON3 has, like other SPARC processors, a branch delay slot: As part of processing any branch instruction, the immediately following instruction is executed first. This behavior has to be considered in our optimization as well.

The LEON3 was chosen as target processor for this paper based on its reasonably well documented instruction timings, especially regarding pipeline stalls [6, Section 83.2.2][7].

3 Design and Implementation

In this chapter we first explore the different instructions timings affecting the instruction pipeline. Then we explain how to mitigate these effects in theory. Finally we show how this theory has been implemented in practice.

3.1 SPARC Instructions and their Pipeline Effects

Most SPARC instruction cause no further pipeline delay, as their result is immediately available for proceeding instructions, thanks to the backtracking paths mentioned above. Some instructions however do cause delay. They can be broken up timing-wise into several groups [6][7]:

Load instructions incur an additional *load delay* when the result of a load is used by following instructions. Depending on configuration parameters set prior to synthesis the load delay may be 1-2 cycles. If the following instruction is a store and the load result is used only for address calculation, no delay is inserted.

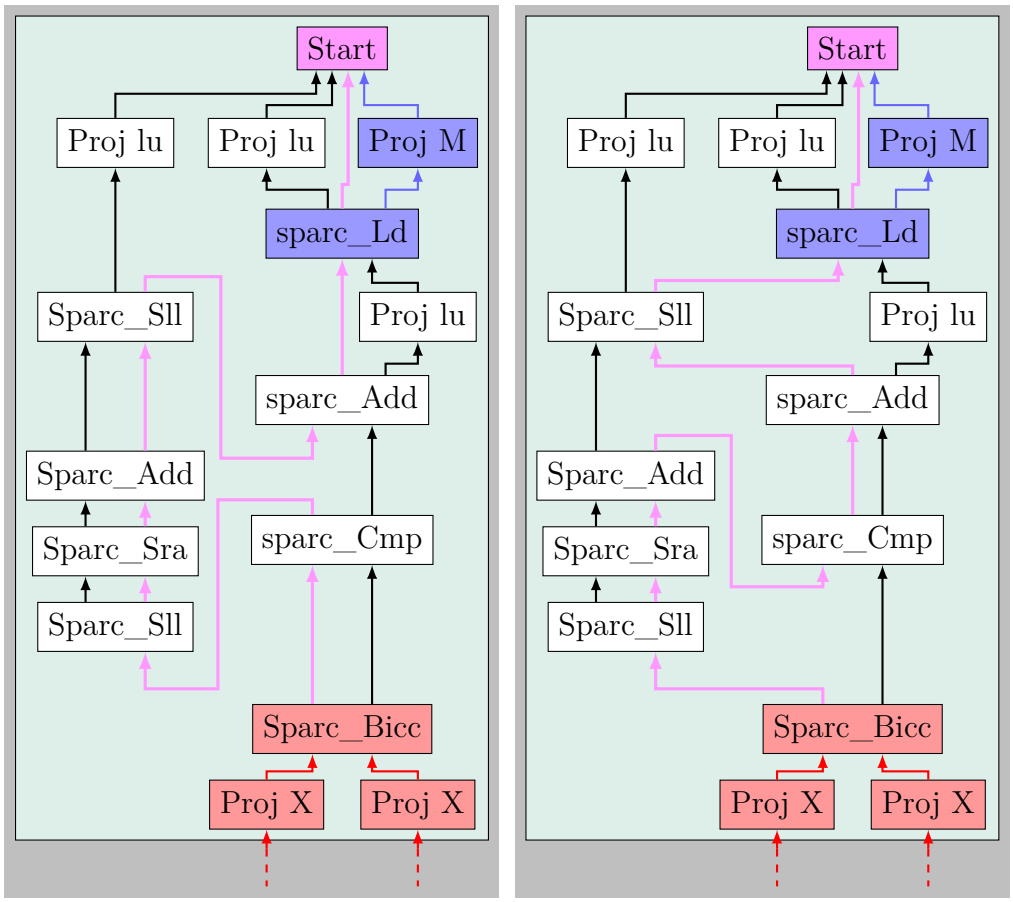
Multiplication requires one cycle of delay if the next instruction uses its result. When used only for address calculation in a store no delay is necessary. The content of the Y register, the most significant 32 bit of the result, can be used without incurring any additional delay.

Division always requires one additional cycle of delay.

Conditional branches can incur additional delay of 1-2 cycles called *branch interlock*. If the immediately preceding instruction modifies condition codes, two delay cycles are inserted, if only the second preceding one does, one cycle.

3.1.1 The delay slot

Additionally the compiler is required to fill the delay slot of any branch instruction. If no suitable instruction is found, a *NOP* is inserted. Scheduling with this in mind, some NOP insertions can be mitigated by placing one such suitable instruction immediately preceding the branch in the scheduling phase, allowing the assembly emitter to fill the delay slot productively.



(a) Conventional scheduling **(b)** SPARC-specific scheduling

Figure 3.1: Sample program exhibiting both *branch interlock* and *load delay*. In the conventional schedule, the branch is scheduled right after compare, forcing the processor to insert 2 cycles of delay. The first Add is scheduled right after the load it depends on. This leads to 1 cycle of load delay. In addition, no instruction to fill the delay slot is found, so a "nop" is inserted. Conventional wastes 4 cycles in total while SPARC-specific scheduling wastes none.

3.2 Design

Scheduling decisions are made on a block-wise basis in libFirm. Mitigating the different kinds of delay requires different approaches:

Branch-related delays can be reduced or eliminated by scheduling ideally up to three instructions between the last instruction that modifies condition codes and the branch of the block. Two of those instructions will fill the time window between compare and branch, allowing the condition codes to propagate properly. The third instruction is later emitted in the branch delay slot, preventing the filling with a NOP. To be suitable, each instruction has to satisfy the following constraints: Firstly, it must not modify condition codes as that would falsify the branch decision. Secondly, it must not be depended on by any node in the current block that is not also in the three nodes selected. Finally, it can not be a predecessor to the last node modifying condition codes, as that would restrict its position to strictly above the same. Any such node can be scheduled directly before the branch, reducing the branch delay and allowing the delay slot to be filled. All other delays can be mitigated on-the-fly by respecting the rules of section 3.1.

In the sample schedule shown in Figure 3.1a both load delay and branch interlock are encountered: The value returned by the *sparc_Ld* is used by the immediately proceeding *sparc_Add* instruction, forcing the processor to insert a bubble in between. The SPARC-specific scheduling however interleaves the first *sparc_Sll* instruction which can be executed at that time since it doesn't depend on the *sparc_Ld*. The resulting machine code can be processed without inserting any bubbles after the *sparc_Ld*, saving one cycle. The same applies to the *sparc_Bicc*: since it follows directly after the *sparc_Cmp*, two bubbles are inserted by the processor. In the improved schedule in Figure 3.1b three instructions are scheduled in between, providing enough time for the condition codes to propagate, saving two cycles. In Addition the emitter might not be able to fill the branch delay slot productively in Figure 3.1a, depending on the available common instructions in the branched-to basic blocks (not shown) that could be pulled up. In the SPARC-specific schedule the emitter is presented with a fitting instruction by scheduling it just before the branch. Overall, four cycles are saved in schedule Figure 3.1b.

3.3 Implementation

The SPARC scheduler is, just as the conventional scheduler, based on the already existing list scheduler. Its interface provides a set of all possible nodes that could be scheduled next, ensuring that all necessary dependencies have already been scheduled prior. This provides any implementing scheduler with some sense of correctness, as any choice made from that set is factually correct, although possibly not optimal. The SPARC scheduling consists of two phases:

The preprocessing phase searches for candidates to resolve branch interlock. This is done through an upwards DFS starting from the branch. Since the DFS visits all

predecessors of the last conditional, any node not visited is not a predecessor and therefore a possible candidate for resolving branch interlock. Three such instructions are chosen, making sure that neither of them violate the rules laid out in section 3.2. Additionally, the total number of nodes is tallied up for later use.

The main scheduling phase performs the actual scheduling. First, each of the choices offered by the list scheduler is scored according to their characteristics in section 3.1: Instructions that are the first node of a hazard-producing pair are scored highly, resulting is earlier scheduling. The second node of a hazard-producing pair is scored negatively, allowing for different, non-hazarding nodes to take the spot, breaking up the hazard. Nodes that modify condition codes (excluding the last one just before the branch) are scored slightly above average. This schedules them slightly earlier which is preferred since they are useless in resolving branch interlock. Any other node is scored neutral. Nodes reserved by the preprocessing phase are excluded from being chosen and will be handled separately. This includes the actual branch and the last node modifying condition codes as well.

Once all non-reserved nodes have been scheduled, branch interlock is resolved by scheduling the remaining, reserved nodes in the correct order. This point is determined through the total number of nodes found by the DFS.

4 Evaluation

In the following section we analyze the the results of the SPARC-specific scheduling and compare it to the conventional scheduling previously used. First, we explain the methodology used in our tests. Then we look at some of the non-resolved hazard-producing dependencies still found by the static analysis, and reason their existence. Finally, we compare the SPARC-specific scheduling versus the conventional scheduling in some dynamic tests.

4.1 Methodology

In order to provide a point of comparison to any dynamic results measured, we employ a primitive static analysis tool written in Python based on the Capstone Disassembly Framework [8]: Starting at the binary entry point, we explore all possible execution paths using a depth-first approach. When reaching a conditional jump, both possible paths are explored. Any jump target is only visited once, preventing infinite looping. Indirect jumps are ignored since static analysis of them is difficult. Our samples do not contain any indirect jumps, so our results below are unchanged by this restriction. While exploring, a list of the last few visited instructions is kept, allowing us to find any hazard-producing dependencies. At the end, the number of them is tallied up by group as in section 3.1 and printed. This method can not give a concrete number of cycles wasted, as determining that statically is hard, it however does provide an overview of opportunities missed. Furthermore we can get a rough estimate of cycles wasted by matching up each missed dependency with its cycle count in section 3.1. This however makes the following assumptions: Firstly no jump target is visited twice, loops are treated as linear code paths only taken once. Secondly, the likelihood of execution is treated as equal for all code paths. These assumptions are obviously violated even by basic programs, the resulting numbers however offer still a meaningful point of comparison.

Dynamic measurements were taken by executing test code under the operating system OctoPOS [9], providing clock cycle accurate time stamping and preventing preemption of our sample. OctoPOS itself ran on an FPGA chip, loaded with a LEON3 synthesis running at 50MHz.

As test programs we used a small, standalone implementation of AES [10] and a selection of samples from the "Computer Language Benchmarks Game" [11], namely *fasta*, *nbody*, *recursive* and *partial*. Each sample was linked with stubbed *printf*, *puts* and *putchar* functions in order to reduce variance in measurements. We chose stubbing instead of removing these function calls from samples to ensure data

dependencies stayed the same and no runtime-contributing parts where optimized out.

4.2 Static Analysis

For our static analysis each sample was compiled once with the SPARC-specific scheduling and once without. Then the number of hazard-producing dependencies left and cycles wasted on them was measured as explained in section 4.1. The results can be seen in Table 4.1. Load, Multiply and Divide Delay are resolved fairly easily by the local prioritization described in section 3.3. Manual inspection of the generated schedule graph confirms this. The branch delay is the most common type of delay still encountered for most samples. This is expected as resolving it is trickier than the other delay forms: Only if other instructions, not participating in the branch-deciding result, are available in the same block, branch delay can be resolved. This however seems less common, as the cparser front end groups dependent instructions together into the same block, seldom leaving a few suitable instructions dangling into the next one. In addition, many non-counting (e.g. while) loops can't be improved since calculations done inside often influence the loop condition, thus becoming unsuitable in resolving branch delay. In general the scheduler can only improve, if there are opportunities to do so: Code samples with tiny basic blocks don't provide interleavable instructions to resolve conflicts. The same goes for large basic blocks which contain only one long dependency chain. If the scheduler is only offered one choice, no improvements are possible.

Table 4.1 hints at most improvement possible for *n-body*, as the difference of total delay between SPARC and conventional scheduling is the largest. This however might be deceiving since static analysis does not account for multiple passes taken in loops. In addition this static analysis provides a sense of maximum improvements possible: If the difference between SPARC and conventional scheduling is small to none, no dynamic improvement can be expected.

4.3 Dynamic Analysis

In our dynamic analysis we compared the actual cycle count of our sample programs with and without SPARC-specific scheduling. This shows the actual time improvement, taking into account the multiple times one piece of code might be executed. The results can be seen in Table 4.2. By stubbing function calls like *printf*, we removed most of the variation in measurements, resulting in acceptable interquartile ranges of $IRQ_{tinyAES} = 8$, $IRQ_{fasta} = 3967$, $IRQ_{n-body} = 21705$, $IRQ_{recursive} = 65$ and $IRQ_{partial} = 43$. Contrary to the results of section 4.2, *tinyAES* exhibits the largest improvement. This can be explained by its internal structure typical for AES, large loops containing multiple passes of long code blocks. Any improvement in one of these blocks gets multiplied by each loop pass taken over in, which leads to an

overall larger improvement. In addition, multiple AES modes are tested by *tinyAES*, amplifying this effect further.

Sample	Delay (cycles)					
	Load	Branch	Delay slot	Multiply	Divide	Total
SPARC-sched.						
tinyAES	8	53	3	0	0	64
fasta	5	14	2	1	0	22
n-body	20	16	2	0	0	38
recursive	4	9	1	0	0	14
partial	15	18	3	0	0	36
conv. sched.						
tinyAES	10	55	5	0	0	70
fasta	9	16	2	2	0	29
n-body	33	16	3	0	0	52
recursive	7	9	3	0	0	19
partial	18	20	6	0	0	44

Table 4.1: Statically found cycles wasted by all hazards grouped by delay type for SPARC and conventional scheduling. This assumes that each hazard is hit exactly once.

Sample	Cycle count scheduled with		improvement in %
	SPARC-specific.	conventional	
tinyAES	213 797	219 520	2.607
fasta	1 047 583	1 057 520	0.940
n-body	30 464 185	30 604 208	0.458
recursive	197 620 158	197 620 149	−0.000
partial	38 910 567	38 771 068	−0.360

Table 4.2: Median cycle count of 1000 runs of each sample, with and without SPARC-specific scheduling

5 Conclusion and Future Work

While SPARC-specific scheduling offers greater improvements than initially expected, all samples tested where rather small. For most samples, improvements made where so small, that even a single *printf* statement in the original code hid any cycles saved in its noise. The abundance of remaining hazards found after SPARC-specific scheduling shows, that a block-based scheduling approach does not offer the flexibility necessary to resolve a large percentage of them. Given that even small programs as those tested exhibited plenty of hazards, inter-block approaches might be able to yield better results. In larger programs register pressure increases and any improvements made are quickly outweighed by cycles lost due to register spilling. Here, a hybrid scheduling approach may increase performance, scheduling like proposed in this thesis while determining register pressure and falling back to the conventional scheduling algorithm in hopes of evading register spilling.

The LEON3 used in this thesis was synthesized with the "GRFPU Lite - IEEE-754 Floating-Point Unit"[6, Section 49], which is not pipelined and can only executes one floating point instruction at a time. It can however alternatively be synthesized with the "RFPU - High-performance IEEE-754 Floating-point unit"[6, Section 49], which is fully pipelined, executing in parallel with the main core. Scheduling with this in mind might offer further room for improvement in float-heavy operations.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] Cobham Gaissler, *LEON 3 Product Sheet*.

[3] M. Trapp, G. Lindenmaier, and B. Boesler, "Documentation of the intermediate representation firm," tech. rep., 1999. Karlsruhe 1999. (Interner Bericht. Fakultät für Informatik, Universität Karlsruhe. 1999,14.).

[4] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[5] "Leon3 product page." `https://www.gaisler.com/index.php/products/processors/leon3`.

[6] Cobham Gaissler, *GRLIB IP Core User's Manual*, Dec. 2019. Version 2019.4.

[7] J. van Rantwijk, "Instruction scheduling for LEON3." `http://jorisvr.nl/article/leon3-insntiming`, Dec. 2015.

[8] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, vol. 5, no. 2, pp. 3–8, 2014.

[9] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat, "Octopos: A parallel operating system for invasive computing," 04 2011.

[10] kokke, "Tiny AES in c." `https://github.com/kokke/tiny-AES-c`, 2020.

[11] I. Gouy, "The Computer Language Benchmarks Game." `https://benchmarksgame-team.pages.debian.net/benchmarksgame`.

Erklärung

Hiermit erkläre ich, Philipp Schaback, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

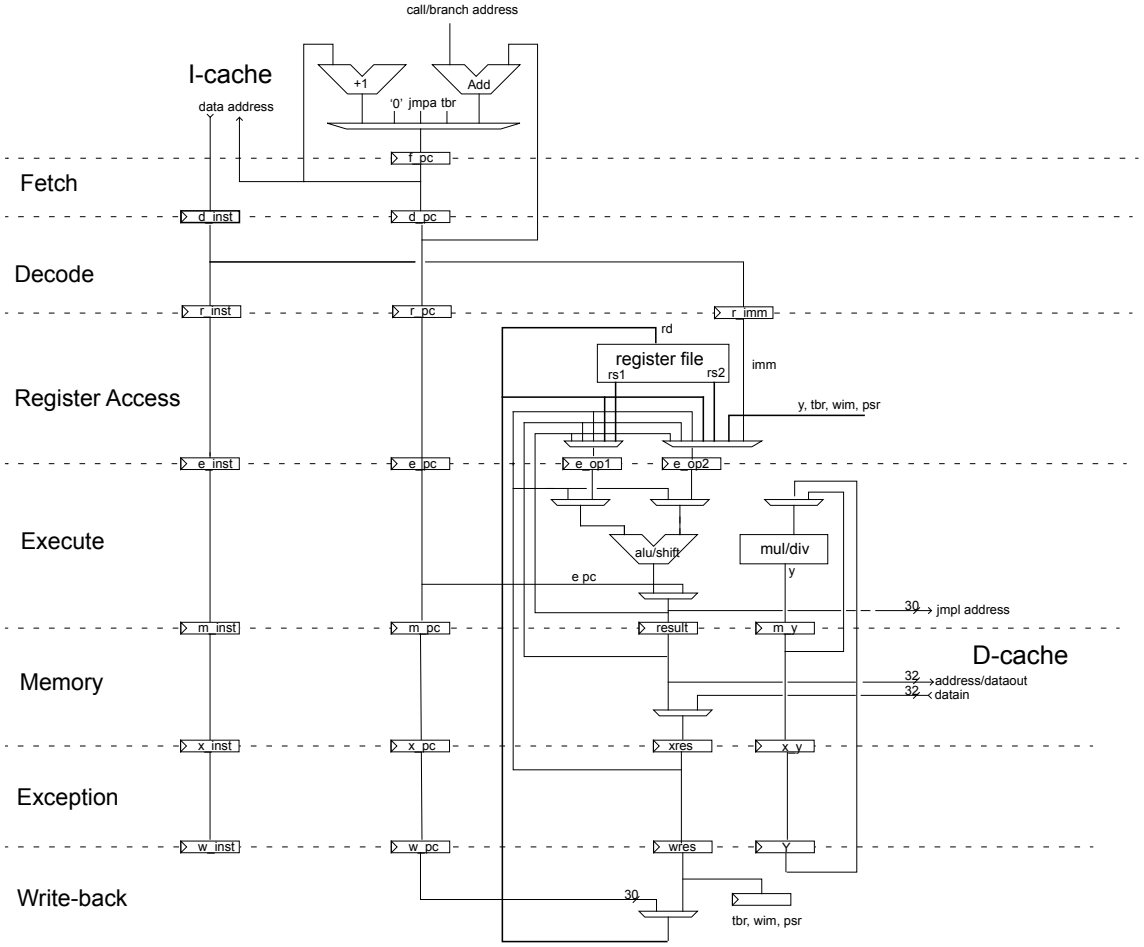


Figure .1: Block diagram of the LEON3 instruction pipeline [6, Section 83.2]