FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Instruction Set Extensions for Time-Predictable Code Execution

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Clemens Bernhard Geyer
Matrikelnummer 0427482

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Wien, 9. Mai 2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Instruction Set Extensions for Time-Predictable Code Execution

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer engineering

by

Clemens Bernhard Geyer
Registration Number 0427482

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Vienna, 9. Mai 2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Clemens Bernhard Geyer
Ohligsgasse 6, 1110 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Nowadays, highly dependable real-time systems are part of many applications in the aerospace and automotive industries. The requirements of real-time applications do not only include the correctness of results, but also the instant of time, when a result is available. In case of a so-called *hard real-time system*, the whole system may crash if a task does not finish within a given period of time. Thus, knowing worst-case execution times of programs in advance is of utmost importance.

Worst-case execution time analysis (WCET analysis) calculates the longest possible duration an application may take to finish. To do so, all external and internal influences have to be considered, for example, processor and memory load, the implemented caching strategy, etc. In order to simplify the analysis of assembler code, Puschner and Burns presented the so-called *single-path transformation* of algorithms in [PB02, Pus03]. It is based on the idea to eliminate conditional branches such that just one possible execution path remains which is trivial to analyze. Nevertheless, this is only possible if the processor provides certain kinds of instructions.

Within the current thesis, the instruction set of the SPARC V8 processor has been extended so that the analysis of assembler code is simplified. Additional goals included that it should be easy to implement these instructions in hardware and adapt existing code generators to support the instruction set extension. Moreover, the resulting worst-case performance should be improved. In order to evaluate the feasibility of the additional instructions, new code generating passes have been added to an existing compiler and an instruction set simulator has been implemented. Based on the results of numerous simulated benchmark algorithms, the most promising instruction set extensions have been identified and suggested to be part of future processors used in real-time systems.

Kurzfassung

Hochzuverlässige Echtzeitsysteme sind heutzutage Teil vieler Anwendungen im Bereich der Luft- und Raumfahrt, sowie der Automobilindustrie. Die Anforderungen betreffen dabei nicht nur die Korrektheit der gelieferten Ergebnisse, sondern auch den konkreten Zeitpunkt, wann diese der Anwendung zur Verfügung stehen. Im Falle eines so genannten *harten Echtzeitsystems* kann es zur Katastrophe, zum Beispiel einem Flugzeugabsturz, kommen, wenn eine Anwendung zu lange für ihre Berechnungen benötigt. Deshalb ist es wichtig, schon im Vorhinein die maximalen Ausführungszeiten eines Programms zu kennen.

Die *Worst-case execution time Analysis* (WCET analysis) befasst sich mit der Berechnung der längst möglichen Ausführungszeiten eines Programms. Dabei müssen theoretisch alle Aspekte eines Prozessors, wie aktueller Speicherzustand, das zu Grunde liegende Caching Modell etc., berücksichtigt werden. Um die Analyse des Assembler-Codes zu vereinfachen, wurde von Puschner und Burns in [PB02, Pus03] eine Transformation vorhandener Algorithmen vorgeschlagen, sodass nur mehr ein möglicher Ausführungspfad existiert. Allerdings unterstützen nicht alle Prozessoren die dazu nötigen Instruktionen.

Im Rahmen dieser Diplomarbeit wurde das Instruction Set des SPARC V8 um Befehle erweitert, die die Analysierbarkeit des resultierenden Assembler-Codes vereinfachen sollen. Weitere Ziele waren eine einfache Umsetzung der zusätzlichen Befehle in Hardware, eine möglichst leichte Integration in vorhandene Codegeneratoren, sowie eine Verbesserung der Worst-case Performance. Um ein möglichst unverfälschtes Ergebnis zu erhalten, wurde ein vorhandener Compiler so angepasst, dass die vorgeschlagenen Erweiterungen bei der Übersetzung berücksichtigt werden. Außerdem wurde ein Simulator entwickelt, sodass die Performance der zusätzlichen Befehle anhand mehrerer Benchmark-Algorithmen erhoben werden konnte. Auf Grund der Messergebnisse war es möglich, die vielversprechendsten Kombinationen der Befehlserweiterungen zu identifizieren und sie als Grundlage für zukünftige Prozessoren im Echtzeitbereich vorzuschlagen.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Examples

List of Abbreviations

ADC	Analog to Digital Converter
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuits
BCET	Best Case Execution Time
CFG	Control Flow Graph
CPI	Cycles per Instruction
CPU	Central Processing Unit
DES	Data Encryption Standard
DMA	Direct Memory Access
DSP	Digital Signal Processor
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
I^2C	Inter-Integrated Circuit
ISA	Instruction Set Architecture
JPEG	Joint Photographic Experts Group
JVM	Java Virtual Machine
LRU	Least Recently Used
LSB	Least Significant Bit/Byte
MAC	Multiply And Accumulate

MBB Machine Basic Block

MCU Microcontroller Unit

MSB Most Significant Bit/Byte

NOP No Operation

PWM Pulse Width Modulation

RAM Random Access Memory

RISC Reduced Instruction Set Computer

SIMD Single Instruction Multiple Data

SPI Serial Peripheral Interface

SRAM Static Random Access Memory

SSA Single Static Assignment

UART Universal Asynchronous Receiver Transmitter

VGA Video Graphics Array

VHDL Very High Speed Intagrated Circuits Hardware Description Language

VLIW Very Large Instruction Word

WCET Worst-Case Execution Time

CHAPTER 1

Introduction

1.1 Problem Statement

Real-time systems have become more and more important over the past decades: In the 1980's, the automotive and aerospace industries started to use digital controllers for several non-critical tasks. Nowadays, nearly all regulating operations of an airplane or a car are executed by embedded systems. For normal computer programs, the main requirement is that they deliver correct results. Time-critical controlling tasks have the additional requirement to deliver correct results until a specified instant of time, which is called a *deadline*. If the whole system may collapse in case the deadline cannot be met, it is called a *hard real-time system*. Examples are controlling tasks in an airplane or a nuclear power plant. If the system is only affected for a short period of time in case the deadline is missed, the system is denoted as a *soft real-time system*, confer [Kop97, p. 2f]. These include, for example, digital telecommunication systems.

One important aspect when dealing with real-time systems is the predictability of the implemented algorithms. If a task may take an arbitrary long time to finish, it could miss a deadline, causing the whole system to crash. Hence, it is of utmost importance to identify *worst-case scenarios* such that the timing constraints are known in advance. Unfortunately, modern processors are not designed to behave predictably, but to speed-up the average case performance. They make use of caches, static and dynamic branch prediction, pipelining, etc. All of these features have to be taken into account for a *Worst-case execution time* (WCET) Analysis. However, it is not always possible to reliably identify the worst-case scenario when all external influences are considered. This is the reason why WCET analysis often uses simplified models of processors, delivering pessimistic results.

For the current thesis, we restrict the focus of WCET analysis only to the influence of the instruction set provided by the processor: Conditional branches as well as instructions with variable execution times increase the number of scenarios to be analyzed. Moreover, the assembler output of the compiler highly depends on the available instructions: If, for example, an algorithm involves a division which is not provided by the instruction set, it has to be emulated by software. Of course, this solution has a much worse performance in comparison with an in-

struction provided by hardware. Thus, the resulting code quality of the assembler output may be improved by supporting additional instructions.

Although there are several approaches defining time predictable processor architectures, none of them analyzes the impact of the underlying instruction set architecture (ISA) on the resulting assembler code. This thesis tries to identify instructions which should be part of such architectures in order to get predictable and easy to analyze assembler code. These instructions should (1) easily be implemented in hardware, (2) easily be added to an existing instruction set, (3) be used by code generating passes and (4) improve the predictability and performance of the resulting assembler code.

In a first step, instruction sets and uncommon features of existing processors have been analyzed. Moreover, an evaluation of various approaches of time predictable processors has been done. In a second step, the SPARC V8 instruction set has been extended by several instructions which meet the previously described requirements. These additional instructions have been added to the code generating pass of the LLVM compiler framework. Hence, the impacts of the instruction set extensions on the assembler output could be verified. To analyze the effect of the newly introduced instructions on the worst-case performance, an instruction set simulator of the SPARC V8 has been implemented. Finally, a big variety of algorithms has been translated to assembler code. By enabling different combinations of the instruction set extensions in the code generating phase, the most feasible solutions in terms of code size and worst-case performance could be identified. The additional instructions which had the most positive influence on performance and WCET analysis have been proposed to be part of instruction sets of future time predictable processors.

1.2 Overview of WCET Analysis

In [PB00], Puschner and Burns define the targets of WCET analysis:

- WCET Analysis calculates *safe upper* bounds of task execution times. This means that the actual worst-case scenario may indeed have a lower execution time.
- WCET Analysis should deliver *tight* bounds of task execution times. Hence, the calculated results should be safe, but not too pessimistic as they otherwise would be useless.

Calculating the worst-case execution time of a given program is not possible for the general case, because it is an instance of the famous *Halting Problem* as indicated by Kirner and Puschner in [KP05]. This is the reason why most WCET analysis techniques try to simplify the problem: By providing additional information such as *flow facts*,[1] an analysis tool is able to identify the worst-case scenario more easily. Another possibility is to ignore the underlying hardware in a first step and to extract flow facts from the control flow graph (CFG) of the high-level code. This information is used later to calculate the exact worst-case cycle count on assembler code level.

As WCET analysis including cache and pipeline models becomes more and more complex, Puschner and Burns introduced the so-called *single-path transformation* in [PB02]. It is based

[1] *Flow facts* are presented in [KP05] and denote descriptions which paths within the control flow graph of a program are to be taken more likely.

on the idea that nearly all algorithms used in real-time systems can be implemented with fixed loop bounds and no complex loop exiting conditions. The minimal precondition, an instruction set has to provide in order to do a single-path transformation, is a conditional move instruction with constant execution time. In case conditional or predicated instructions are supported by the instruction set of the processor (see, e.g., Section 2.2.4), it is simpler to apply the single-path transformation. The main idea is to execute both, the if- and else-branch, and to decide afterwards which result will be used. It is also possible to transform nested if-then-else structures and loops with fixed bounds as presented in [PB02, Pus03]. Thus, the analysis of the corresponding control flow graph is trivial because only one possible path exists. Nevertheless, the resulting code usually shows worse performance, not only in the average, but also in the worst case.

1.3 Outline

Chapter 2 gives an overview of existing state-of-the-art processors used in embedded systems. The main focus lies on the provided instruction set and tries to evaluate whether it fully supports the single-path transformation. Nearly each of the presented processors features special instructions. It will be evaluated whether these instructions are useful in order to simplify WCET analysis. Chapter 3 presents the most interesting scientific approaches which try to define a time predictable architecture. Some of them have really been implemented in hardware, whereas others only enumerate basic concepts.

The main part of the thesis can be found in Chapters 4 to 6. First, the SPARC V8 processor and suggestions for instruction set extensions are presented. The chapter also includes opcode considerations and possibilities of how the extensions might be implemented in hardware. Chapter 5 gives an overview of the LLVM compiler framework and how it has been extended to support code generation for the additional instructions. The performance evaluation of the instruction set extensions is done in Chapter 6. It is based on a number of benchmark algorithms, which have been translated into multiple assembler codes: One implements the original SPARC V8 instruction set, others provide different combinations of the proposed instruction set extensions. Based on the resulting code sizes and worst-case performance, two combinations of additional instructions have been decided to be most feasible for processors being predictable and easy to analyze.

CHAPTER 2

Instruction Set Analysis

This chapter gives an overview of common state-of-the-art embedded processors/microcontrollers[1]. The focus lies on the instruction sets they provide, but additional information about important hardware features is also presented. The evaluation of the processors is mainly based on the impacts on WCET analysis, i.e., whether the ISA and properties of the individual processors provide useful functionality in the domain of real-time systems.

All presented microprocessors are used in different types of embedded applications, most of them related to real time systems such as controllers in the automotive industry or mobile devices in the field of telecommunications. Of course, the selection may not cover all available processors, e.g., the commonly used PowerPC is not part of the current analysis because it does not have any features which are specific to real-time systems or which are not covered by the other presented processors.

The aim of the current chapter is to present instructions or hardware features which are unique for a single processor and may be useful extensions for a time-predictable architecture. Consequently, only features which have not been described so far are explained in full detail. The order of presentation is alphabetical, based on the name of the manufacturer, and is no indication of the quality or importance of the individual processors.

2.1 Analog Devices Blackfin Microprocessor

2.1.1 General Features[2]

According to [Ana11], Blackfin processors are widely in use for embedded audio and video applications, voice and image processing, in the field of real time security, and in other applications. As the Blackfin is mainly used in signal processing, it combines features of a typical DSP with the possibilities and flexibility of microcontrollers. This may be the reason why Analog

[1] Although these terms usually carry different meanings, there is no generally valid definition. Thus, *processor* and *microcontroller* are often used equivalently, depending on the specific definition of the manufacturer.

[2] Confer[GK07, p. 163-167], [Ana08, p. 1.1-1.6] and [Ana10, p. 1.1-1.10].

Devices declares the Blackfin an *embedded processor*. It is based on a RISC architecture with an instruction size of 32 bits. However, frequently used instructions are encoded with 16 bits.

The Blackfin processor allows to address up to 4 GByte of internal and external memory, which are used for caches, internal RAMs and I/O registers. 4 KByte of the internal L1 memory may be used as a scratchpad such that time-critical applications get guaranteed access times to requested data. The processor has eight 32-bit registers `R0` to `R7`, which may also be used as sixteen 16-bit registers (e.g., by separately accessing the lower and upper 16 bits of the corresponding registers `R0.L` and `R0.H`). There are six 32 bit pointer registers which are used for address calculations. Moreover, the Blackfin has two independent 40-bit accumulators, which support multiple computational operations[3] for a single instruction (see Section 2.1.3 for details). Of course, there are several more registers, but they are mostly for internal usage or special instructions.

As already mentioned, the Blackfin processor is not mainly designed for applications in hard real-time systems. Nevertheless, it offers some instructions supporting the WCET analysis of algorithms commonly in use in such kind of systems. In the following sections, these instructions are presented, together with instructions which are rather uncommon and specific to the Blackfin processor.

2.1.2 Instructions for Calculating Minimum and Maximum Value[4]

The Blackfin processor offers the programmer hardware instructions to determine the minimum and maximum of two given integers. The given input numbers are interpreted as signed 16- or 32-bit values, depending on the instruction type. A number is said to be negative if the MSB is set. It is also possible to evaluate both operations independently on two 16 bit registers (see Section 2.1.3).

The application of these instructions is mainly to provide a hardware implementation of the `MIN()` and `MAX()` functions, which are used in nearly all high-level programming languages[5]. It may help to find the smallest/greatest element of an unsorted array in shorter time because the integer comparison is done in hardware and saving the smaller/greater value does not require additional conditional branches. Consequently, the control flow graph of the program gets simpler and WCET analysis is much easier.

2.1.3 Single Instruction Multiple Data (SIMD)[6]

The Blackfin processor offers several possibilities to perform multiple operations with just one instruction. In principle, there are two possibilities:

[3] In the current thesis, the term *instruction* refers to an assembly or high level language instruction whereas *operation* is equivalent to a basic mathematical function implemented in hardware. Thus, an instruction may unite multiple operations; the opposite is usually not true. In some cases the terms may be used interchangeably, e.g., in context of arithmetic instructions or operations.

[4] Confer [Ana08, p. 15.31-15.35].

[5] Usually, in the C programming language a macro like `#define MAX(a,b) ((a)>(b))?(a):(b);` is defined for this purpose.

[6] Confer [GK07, p. 167-173] and [Ana08, p. 19.1-20.10]

(1) An instruction performs two or more operations on one 32-bit register, but the lower and upper 16 bits are treated separately. Analog Devices calls this type of instruction a *vector operation*.

(2) There are certain instructions which can be performed in parallel, beginning with one 32-bit instruction followed by two 16-bit instructions. These three are treated as one 64-bit instruction. Analog Devices calls this type a *parallel instruction*.

Additionally, the Blackfin processor offers so-called MAC instructions (Multiply And Accumulate). They allow to add or subtract the result of a multiplication to or from the currently saved value in one of the accumulators. Although this single instruction performs multiple operations, it is a quite common feature of processors implementing an accumulator architecture. Moreover, it is a typical feature of DSPs, which allows the implementation of efficient matrix-matrix multiplications, which are needed for Fourier transformations. See [Ana08, p. 15.56-15.70] for a detailed description.

Vector operations usually work on 16-bit registers. However, two 32-bit registers may also serve as vector input for arithmetic instructions. Code example 2.1 shows different possibilities for this kind of operations: The first line subtracts the upper 16 bits of `r3` from the upper 16 bits of `r2`, while at the same time the lower 16 bits of each register are added. The next example shows how two 32-bit registers are simultaneously added and subtracted while the results of both operations are saved to different destination registers. The last code line performs four arithmetic instructions with eight 16-bit input registers.[7] The four results of these operations are saved in the upper and lower 16 bits of registers `r0` and `r1`, respectively. Note that each *real* instruction is closed with a semicolon, whereas two instructions belonging together are separated by a comma.

Code Example 2.1 Examples for arithmetic vector operations of the Blackfin processor.

```
r0 = r2 -|+ r3;
r3 = r6 + r7, r4 = r6 - r7;
r0 = r2 +|- r3, r1 = r4 -|- r5;
```

There also exist vector operations for the simultaneous calculation of the minimum and maximum of two 16-bit values. In contrast to arithmetic instructions, only one of the two operations may be performed on both registers. Moreover, vector multiplications as well as vector MAC instructions are supported. They will not be discussed here as the principle is the same as for the instructions just presented.

Parallel instructions have a special syntax as it is only possible to execute certain operations concurrently. The first instruction has to be a 32-bit instruction and may also be a NOP. Supported are arithmetic, move, bit operation and vector operation instructions. It is followed by two 16-bit instructions which are subject to certain restrictions. For example, it is not possible to access registers used by the first 32 bit instruction. Nevertheless, it is possible to perform a MAC

[7] The involved source operands are the upper 16 bits of `r2` and `r3`, the lower 16 bits of `r2` and `r3` as well as the lower and upper 16 bits of `r4` and `r5`.

operation and execute two read or write instructions in the same instant of time. The so-called *Instruction Alignment Unit* of the Blackfin processor always accesses eight bytes from the instruction memory to ensure that all instructions are aligned properly. This is necessary because there are 16-, 32- and 64-bit instructions. The last format is used for parallel operations. Consequently, the instruction fetch operation always takes the same amount of time, regardless of the current bit width. In case of a parallel instruction, the execution duration of each of the three instructions is equal. See [Ana08, p. 4.7-4.9] for further details about the pipelining concept of the Blackfin processor.

Beside the functions presented in the current section, there are so-called *video pixel operations*. They are also typical for DSPs and allow for the simultaneous calculation of the pairwise average of eight bytes. This feature may be useful when calculating a transition image of two given pictures. See [Ana08, p. 18.1-18.40] for more details about this kind of instructions.

2.1.4 Instructions for Cache Control[8]

As already mentioned in Section 2.1.2, the Blackfin processor allows the programmer to define the usage of its L1 memory. This is done by writing configuration bits into the `DMEM_CONTROL` register, indicating whether some parts of the memory should be used as an instruction cache, a data cache, as usual SRAM or as a scratchpad. Although the presented instructions are not very powerful on their own, it is a quite uncommon feature to let the user perform cache-related operations, which are usually only implemented by a hardware controller.

When L1 is defined as a data cache, the programmer has the possibility to perform some actions to manipulate the content of the cache. This may help to provide an easier analysis of execution time or to guarantee worst-case timings. The `prefetch` instruction allows the user to load a memory region into the internal data cache. The address of that region is provided by a pointer register. The `flush` and `invflush` operations provide the possibility to explicitly perform a read from or write to higher memory levels if the cache line has been marked as *dirty*.

2.1.5 Hardware Loops[9]

The Blackfin processor allows the programmer to make use of a loop implemented in hardware. This is done by saving all necessary information for the loop bounds in special registers: `LCx` holds the value of the *loop counter*, `LTx` holds the address of the first instruction within the loop, and `LBx` holds the address of the first instruction after the loop. As the processor also supports nested loops for one inner and one outer loop, the registers with suffix "1" (e.g., `LC1`) have higher priority and should be used for the inner loop, whereas registers with suffix "0" can be used either for a single or for the outer loop. The instructions within the hardware loop may be of any kind, but must not be conditional branch, jump or call instructions. Otherwise, the execution behavior of the loop is undefined.

The assembler language definition of the Blackfin processor gives the user three possibilities for implementing the loop: The first five lines of Code Example 2.2 make use of the `loop` - `begin_loop` - `end_loop` construction, which is the most readable form. Note that the three

[8] Confer [Ana08, p. 17.1-17.10] and [Ana10, p. 2.1-2.6]

[9] Confer [GK07, p. 178-182] and [Ana08, p. 7.13-7.19]

instructions in line four are executed in parallel, as indicated by the two vertical bars (`||`) which conjunct parallel instructions presented in Section 2.1.3. The second possibility is also shown in Code Example 2.2 and is semantically equivalent to the upper assembler code: In both cases, the loop counter `LC0` is initialized to "32", `LT0` is set to the address of the first instruction within the loop, and `LB0` to the address of the first instruction after the loop (the last line of each example). It is not necessary to start the loop straight after the `loop` or `lsetup` instruction, which is the reason why the user explicitly has to state the starting address of the loop. `LTx` and `LBx` are set automatically by the two instructions. The end address label is encoded with 11 bits by the instruction, meaning that the end address may be 2046 bytes away from the current program counter at maximum. Consequently, the address range of a loop is quite limited. Nevertheless, it is also possible to manually initialize all needed loop registers such that the address range includes the whole memory of 4 GBytes.

Code Example 2.2 Two possible realizations of hardware-supported loops implemented with the instruction set of the Blackfin processor. The code is based on Example 5.15 in [GK07, p. 181].

```
p5 = 0x20;
loop MYLOOP lc0 = p5;
loop_begin MYLOOP
r2 = r0 + r1 || r3 = [p1++] || r4 = [i1++];
loop_end MYLOOP
r2 = r3 + r4;

p5 = 0x20;
lsetup (loop_start, loop_end) lc0 = p5;
loop_start:
r2 = r0 + r1 || r3 = [p1++] || r4 = [i1++];
loop_end:
r2 = r3 + r4;
```

2.1.6 Conclusion

Although the Blackfin processor is not primarily designed for hard real-time tasks, its instruction set offers quite good possibilities to design WCET-aware applications. The analysis of the CFG can be dramatically simplified by using features like hardware loops, which guarantee predictable timings also in case of nested loops. Moreover, it is possible to influence the cache semantics or even to turn off the cache and use the available memory as a normal RAM. As a cache is always a source of unpredictability, these features also help to improve the correctness of WCET analysis.

Nevertheless, the Blackfin does not provide any conditional instructions like predicated instructions or conditional moves. Thus, it is not possible to apply the single-path approach as described in Section 1.2 to programs optimized for the Blackfin processor. Moreover, due to the 10-stage pipelining concept paired with numerous possibilities of combining multiple instructions (vector operations and parallel instructions) with different bit widths, the timing analysis

of scenarios with multiple possibilities for branch prediction may be very complex or even impossible. Consequently, the Blackfin processor may be an applicable choice for soft real-time systems, but it is not ideal for hard real-time systems with complex timing constraints.

2.2 ARM Processors

2.2.1 General Features[10]

The ARM company was originally formed as a joint venture between Acorn Computers, Apple Computers and VLSI Technology in 1990, but is an autonomous business today. It specializes in designing processors and embedded processors for all types of applications, including smartphones and mobile devices, automotive braking systems and smart sensors, as well as many others.[11] The main goals of the first ARM processors, which were designed as RISC processors, included high performance, high code density, low power consumption, and small die area.

The ARM instruction set uses a constant width of 32 bits, but it is also possible to change to the so-called *Thumb instruction set* (see Section 2.2.5), which only uses 16 bits. This provides an even higher code density, resulting in less power consumption because fewer memory fetching operations for instructions are necessary. In the latest version of the ARM instruction set, it is also possible to switch between the two instruction sets quite easily. Over the years, the ARM instruction set has been extended to include DSP operations and management instructions for handling multi-threaded operations.

ARM processors provide different modes (user, supervisor, system, etc.) with individual register sets. In each mode, there are 17 32-bit registers available (including the program counter and the processor state register), but up to eight registers may be dedicated to a certain mode. This kind of registers are called *banked registers*, whereas the registers which are visible to all processor modes are called *unbanked*.

Current ARM processors make use of the ARMv7 instruction set, although ARMv6 and ARMv5 are still in use. Moreover, the newer versions try to provide backward compatibility so that most ARMv5 instructions are also supported by ARMv6 and ARMv7. In the following subsections, the special features of the ARM instruction set are presented. They include predicated instructions, SIMD instructions, which are partially identical to the Blackfin instruction set, and special operations on floating point numbers. Note that the particular versions of the instruction sets will not be considered because the main focus lies on architectural principles and supported features.

2.2.2 Single Instruction Multiple Data (SIMD)

Like the Blackfin processor, the ARM instruction set provides some features which are quite typical for DSPs. There are instructions allowing the programmer to split two 32-bit registers into four independent 16-bit registers and calculate the sum or difference of them. The simultaneous calculation of two different operations (i.e., add and subtract) on the 16-bit registers is

[10] Confer [GS05] and [ARM05, p. A2.1-A2.15].

[11] Confer [ARM11a] for further information.

also possible. Moreover, the user may split a 32-bit register into four 8-bit registers. In this case, only one arithmetic instruction may be executed simultaneously on all registers.[12]

Besides these types of simple arithmetic functions, it is also possible to multiply two 16-bit values, add or subtract a third register and save the result in a destination register. Alternatively, there are instructions dealing with four 16-bit registers, which are multiplied (separate multiplication of high halfword and low halfword of each 32-bit register). The results of both multiplications are summed up and added to the value of another register. The result of the whole operation is saved to the specified destination register.[13]

So far, the SIMD instructions of the ARM processor nearly provide the same functionality as the instruction set of the Blackfin processor. Nevertheless, there are special load and store instructions allowing to get or save multiple registers from or to memory. The user has access to all visible general-purpose registers of the current mode, including the program counter.[14] Code Example 2.3 shows how concurrently storing and loading multiple registers is done: The first part is the instruction itself, `STM`, which stands for *store multiple*. It is followed by the addressing mode *decrement after*, `DA`, and the base address register `R1`. The optional exclamation mark indicates that the value of `R1` should be updated after the instruction. This might be useful when using `R1` as the base address register of a stack. The last part of the instruction is a list of registers which shall be transferred to the memory. The `LDMIB` (*load multiple with increment before*) instruction in the last line of the example restores the state of the previously saved registers.

Code Example 2.3 Storing and loading of multiple registers with a single 32 bit instruction. The example is based on [ARM05, p. A3.26].

```
STMDA   R1!, {R2, R5, R7 - R9, R11}
...
LDMIB   R1!, {R2, R5, R7 - R9, R11}
```

2.2.3 Floating Point Instructions[15]

Some ARM processors provide operations on floating point vectors, which are executed on a coprocessor. This extension is called the *vector floating-point architecture*, abbreviated VFP. The format of the numbers is based on the IEEE 754 standard and offers single and double precision operations. Besides the usual arithmetic operations (addition, subtraction, multiplication and division), the VFP module also provides a *multiply and accumulate* instruction on floating point registers. One special arithmetic instruction is the *square root* operation, which allows the calculation of the square root of a single or double precision floating point register.

[12] Confer [ARM05, p. A3.14f].
[13] Confer [ARM05, p. A3.10ff].
[14] Confer [ARM05, p. A3.26f, A4.36-A4.42].
[15] Confer [ARM05, p. C1.1-C4.126]

2.2.4 Predicated Instructions[16]

One of the main features of the ARM processors are the so-called *predicated* or *conditional* instructions. This means that the programmer can, for nearly every instruction, specify a condition on which it shall be executed. If the condition is not met, the instruction behaves like a NOP. The condition flags for each instruction are evaluated (e.g., equal or less than) and compared with the `CPSR` (Current Program Status Register). This mechanism implies that all instructions which might change the flags of the `CPRS` should also only be able to write them when specified. Otherwise, the condition for all following instructions might become invalid.

The advantage of this technique is that small blocks of if-then-else code can be translated easily. The condition is used as a predicate for all instructions of the then-block, whereas the negated condition serves as predicate for the else-branch. Of course, it is not easy to translate complex nested if-then-else-statements without any conditional branch, but the CFG of the resulting assembler code may be much simpler for the WCET analysis. Moreover, predicated instructions allow the user to convert the CFG of a given program to a single-path variant as explained in Section 1.2.

Code example 2.4 shows a possible translation of a simple if-then-else statement: In the first line, `R3` is subtracted from `R2` and the result saved to `R1`. The suffix `S` after the `SUB` instruction indicates that the flags in the `CPSR` should be updated. The next instruction, which adds `R6` to `R5` and saves the result in `R4`, will only be executed if `R2` and `R3` are equal, meaning that `R1` is zero. The last `ADD` instruction is only executed when `R2` and `R3` are not equal. It is also possible to use any other condition for the instructions, they do not have to be mutually exclusive.

Code Example 2.4 Example for conditional execution of the `ADD` instruction: depending on the result of the first subtraction, either the second or the third line is executed. Confer [ARM11b, p. A8.24f, p. A8.422f].

```
SUBS    R1, R2, R3
ADDEQ   R4, R5, R6
ADDNE   R4, R5, R7
```

2.2.5 Thumb Instruction Set

> "*The Thumb instruction set was developed as a 16-bit instruction set with a subset of the functionality of the ARM instruction set. It provides significantly improved code density, at a cost of some reduction in performance.*"[17]

This quotation shows the original intention of the Thumb instruction set, namely to provide a reduction of the quite powerful ARM instruction set in order to achieve a higher code density. As most ARM processors support both types of instruction sets, it is possible to switch between them at runtime. There are two instructions providing this switching functionality:[18]

[16] Confer [ARM11b, p. A4.3, p. A8.8].
[17] See [ARM11b, p. A1.3]
[18] Confer [ARM05, p. A4.16-A4.20].

`BX` The so-called *branch and exchange* instruction conditionally performs a jump to the address given in the specified register. The LSB of the address register indicates whether the instruction at the destination address is a normal ARM instruction or a Thumb instruction.

`BLX` The *branch, link and exchange* instruction is one of the few instructions which have no conditional field and are always executed. The semantics are quite similar to the `BX` instruction, but the address of the following instruction is saved in `R14`. The instruction can be used for a function call to a subroutine provided in the Thumb instruction set. Returning from the subroutine may be done using the `BX` instruction with `R14` as address register.

Version 2 of the Thumb instruction set was introduced with the sixth version of the ARM instruction set (ARMv6). It provides additional features and also makes use of 32-bit instructions. One major advantage of the Thumb instruction set version 2 is that it supports nearly the full functionality of the ARM instruction set, while still offering a higher code density. Although one of the key features of the ARM instruction set, namely the conditional execution of instructions as presented in Section 2.2.4, is not supported, it is possible to conditionally execute a block of four instructions: Code example 2.5 shows the syntax and semantics of the `IT` instruction, which is an acronym for *if-then*. The single argument of the instruction is the condition for the the subsequent instruction. This is "not equal" (`NE`) in our case. The mask `TTE` after the instruction stands for "then", "then" and "else", meaning that the second and third instruction will also be executed if the condition is met, whereas the fourth instruction is only executed if the condition is not fulfilled. Of course, it is possible to specify any other execution mask. By regularly making use of this instruction, it is possible to provide predicated instructions for the Thumb instruction set, too. One disadvantage is that none except the last instruction may be a jump or similar instruction, because it could provoke undefined behavior according to the manual.

Code Example 2.5 Conditional execution of Thumb instructions by making use of the `IT` instruction. See [ARM10, p. A7.277f] for more details.

```
ITTTE   NE
STR     R2, [SP]
ADDC    R1,R1,#1
LDR     R2, R1
SUB     R1,R1,#1
...
```

2.2.6 Conclusion

ARM processors are currently among of the most widely used processors in embedded applications and provide low power consumption, high code density and a powerful instruction set. Moreover, with the powerful features of floating point operations (e.g. square root implemented in hardware) and predicated instructions, WCET analysis of programs may be easier, especially when the algorithms involve floating point operations presented in Section 2.2.3. Additionally,

ARM processors offer a considerable range of SIMD instructions, which are useful for finding an efficient solution for DSP applications.

Nevertheless, a compiler trying to find an assembler translation feasible for WCET analysis (e.g., a single-path transformation) of a given program requires sophisticated and complex techniques. That is the reason why the programmer has to be aware of the specific challenges and problems of an algorithm and may have to support the compiler.

Another problem regarding WCET analysis is to find the exact timing of instructions: As most ARM processors make use of static and dynamic branch prediction, but only within certain bounds, the calculation of the exact duration of a jump or conditional branch is nearly impossible. Although most operations take one or two clock cycles to finish, there are some instructions, like multiplications and coprocessor instructions, which may take up to five clock cycles, but may finish earlier, depending on the instruction length and whether they are conditional. Consequently, WCET analysis of a concrete single-path translation of a given code may also be very complex, even if the timing model does not include caches, which are also part of most ARM processors.[19]

2.3 Atmel AVR Microcontrollers

2.3.1 General Features

Atmel offers two different types of microcontrollers: One for small embedded applications such as automotive and peripheral controllers, and the other one for general-purpose applications including DSPs. The first type is the 8-bit variant of AVR microcontrollers, the second the newer 32-bit variant. Because the instruction set of the 32-bit AVR was developed completely from scratch, there are not many similarities to the 8-bit version. The naming "8-bit" and "32-bit" only relates to the bit-width of the general-purpose registers and has nothing to do with the instruction length of the instruction sets. In the following subsections, both types of instruction sets and the main features of the corresponding microcontrollers are presented.

2.3.2 8-Bit AVR Instruction Set[20]

Atmel's 8-bit microcontrollers are mainly used for embedded controlling tasks, including distributed systems, and provide programmers with a small, but powerful instruction set. They were introduced by Atmel in 1996 and have been quite successful since then.[21] Nearly all 8-bit AVR microcontrollers provide numerous features for digital and analog input and output. These include analog comparators, ADCs, various timer functions (PWM, input capture, etc.), interrupt-triggered input pins and hardware implementations of different communication interfaces (e.g., UART, SPI, I^2C).

One of the main advantages of the 8-bit AVR instruction set is its simplicity, which makes it quite easy for beginners to get in touch with assembler and microcontroller programming.

[19] Confer [ARM09, p. 16.1-16.34].
[20] Based on [Atm10].
[21] Confer [Atm11, p. 2]

Moreover, most of the microcontrollers are very cheap and affordable for private persons. These are the reasons why they are a common solution for hobby engineers. Nearly all instructions have a length of 16 bits, with the exception of calling, jumping, loading and storing functions with absolute addresses. As all registers have a width of 8 bits, it is possible to initialize a register just with one instruction, which is usually not the case for most instruction sets. The 8-bit AVR instruction set provides all arithmetic functions except for integer division. Moreover, bit manipulating instructions like setting or clearing a bit of a register, data transfer functions like loading from and storing to memory, and control flow instructions like conditional branches are supported. Naturally, all arithmetic functions are also available for integers with more than 8 bits (add with carry bit etc.).

The 8-bit AVR instruction set does not provide any sophisticated SIMD instructions like Blackfin or ARM processors, and there is no possibility of conditional execution of any instruction.[22] Nevertheless, some devices offer a hardware implementation of DES having nearly constant execution times of one encryption or decryption operation. This can make WCET analysis of applications involving DES quite easy, although this is a rather uncommon scenario.

As 8-bit AVR processors do not have any caches or complex pipelines and only use static branch prediction, the timing of all instructions is nearly independent of any constraints. This makes the 8-bit AVR instruction set very convenient for WCET analysis. Nevertheless, there are no features for single-path conversion. Consequently, the WCET analysis is highly dependent on the complexity of the CFG of a given program.

2.3.3 32-Bit AVR Instruction Set[23]

Like the Thumb version 2 instruction set of ARM processors (see Section 2.2.5), commonly used instructions have a length of 16 bits whereas more powerful, but rare instructions are encoded with 32 bits. Like nearly all modern processors, 32-bit AVR microcontrollers offer several processor modes for exception and interrupt handling. Moreover, there are several pipeline stages which allow to execute an instruction nearly every clock cycle. Some 32-bit AVR processors also offer hardware support for the Java virtual machine so that Java byte code can be executed directly, without retranslating the program.

Like ARM and Blackfin processors, 32-bit AVR microcontrollers provide a range of different SIMD instructions, including *multiply accumulate* and parallel adding and subtracting on two halfwords of a 32-bit register (see Section 2.1.3 for further details). Additionally, it is possible to load and store multiple registers from and to memory. Refer to the architecture manual ([Atm11]) for a detailed description of all SIMD instructions.

In contrast to ARM processors, it is not possible to execute all instructions conditionally. Nevertheless, the most common operations may be predicated so that they are only executed when a given condition is met. These operations include conditional add, subtract, return, load and store instructions, which allow single-path conversion of any given code.

Like the Blackfin processor, 32-bit AVR microcontrollers allow to control the cache explicitly: There are instructions in the supervisor mode for prefetching data and instructions and for

[22] One exception are the `sbrc` and `sbrs` (**s**kip next instruction if **b**it in **r**egister is **c**leared, resp. **s**et).

[23] Based on [Atm11]

setting control bits in the page table of the memory. Thus, an operating system can implement a custom replacement strategy for pages and caches.

Another interesting feature is the possibility to evaluate the current performance. The hardware mechanism to do that works as following: There is one counter register which is incremented every clock cycle. Additionally, there are two so-called *performance counters* which are incremented when a specified event occurs. By monitoring certain events, like the completion of an instruction execution or a cache miss, it is possible to calculate performance statistics during operation, with only minimal additional delay.

2.3.4 Conclusion

Although the 8- and 32-bit versions of the AVR instruction set are very different, they both offer some interesting features which could help making WCET analysis easier: On one hand, the timing analysis of 8-bit AVR instructions is easy because there are nearly no internal or external dependencies such as pipeline stalling, cache misses or data dependencies. On the other hand, 32-bit AVR microcontrollers offer some conditional instructions which make single-path conversion possible. Moreover, calculated theoretical results of WCET analysis may be verified by evaluating the performance counter registers.

Unfortunately, the 8-bit AVR instruction set does not provide any predicated instructions, which can make WCET analysis quite complicated when analyzing translated assembler code. The 32-bit AVR is not 100 percent suitable for hard real-time tasks, either: Although a single-path conversion is theoretically possible, the resulting code might be useless for real-time systems due to the increased runtime. Moreover, the provided performance-enhancing features like caches and pipelining prohibit a completely correct WCET analysis.

2.4 Infineon TriCore Processors

2.4.1 General Features[24]

The Infineon TriCore processor was designed with the intention to unite the features of three different types of processors:

- CPUs provide hardware mechanisms which are useful for operating systems. Moreover, CPUs usually have the ability to deal with floating point numbers and support memory management.
- MCUs are efficient when dealing with interrupts and bit manipulations.
- DSPs are designed to provide typical arithmetic instructions for signal processing applications, including multiplication and accumulation.

The TriCore tries to unite the advantages of all three types of processors in a single core, which is the reason for its name *TriCore*. It is mainly in use in automotive applications for control and communication tasks. Although Infineon states in the manual that the instruction set of the

[24] Confer [Inf03a, p. 15-27]

processor is rather large, all instructions are encoded with 16 or 32 bits, which is a common technique for 32-bit processors.

The processor provides 16 data and 16 address registers with 32 bits length. That consequently means that a specific register may only be used by a subset of the instruction set. Nevertheless, operations dealing with address registers are optimized for address calculation, whereas instructions on data registers provide DSP and bit manipulating features.

2.4.2 SIMD Instructions

Like all presented 32-bit processors, the TriCore supports multiply add, multiply subtract and multiply accumulate instructions. Additionally, it also supports the execution of so-called *packed* multiply add and subtract instructions: The lower and upper 16 bits of two registers are multiplied and the 32-bit result of each multiplication is added (subtracted) to (from) a 64-bit register. The concurrent execution of a multiply add and a multiply subtract instruction is also possible.[25]

Nearly all arithmetic instructions (except for division) are also available as *packed* instructions, which means that the operation on a 32-bit register behaves like an operation on two 16-bit or four 8-bit registers. In contrast to the Blackfin processor, simultaneous adding and subtracting of two 16-bit values is not possible. Nevertheless, the TriCore provides packed versions of a minimum and maximum instruction, which allow the calculation of the minimum/maximum of up to four bytes in one instruction.[26]

2.4.3 Conditional Instructions and Branches[27]

Like the 32-bit AVR, the TriCore is not fully predicated, but provides some instructions which are only executed on a provided condition. The condition is saved in a register which may only be tested for equality or non-equality to zero. The TriCore has a 32-bit *select* and a 16-bit *conditional move* instruction. The *select* instruction saves the value of the first register to the destination register if the condition register is zero, otherwise, the value of the second register is saved to the destination register. This behavior can be compared to a hardware multiplexer which selects one of two input signals depending on a control signal. The *conditional move* is a restricted version of the *select* instruction and is only capable of taking data register 15 as conditional register. Moreover, depending on the value of `d15`, either the value of a specified register is saved to the destination register or the destination register remains untouched.

Beside the instructions just presented, there are also conditional *add* and *subtract* instructions. Depending on the value of the conditional register, which is only tested for non-equality to zero, the sum or difference of two registers is saved to a destination register. If the condition register is zero, the value of the destination register remains untouched. There is also a 16-bit version of these instructions: Data register 15 is implicitly used as condition register, and the destination register has to be equal to the second source register of the addition or subtraction.

The Infineon TriCore, like the Blackfin, provides a hardware support for loops: There is one instruction which increments or decrements a register, checks the value, and performs a jump to

[25] Confer [Inf03a, p. 203-215 and p. 232-242]
[26] Confer [Inf03a, p. 30-39 and p. 52f]
[27] Confer [Inf03a, p. 68ff and p. 168ff]

a specified address if the result is equal to zero. Its assembler acronyms are `JNEI` and `JNED`, which stands for **j**ump if **n**ot **e**qual and **i**ncrement or **d**ecrement, respectively. An improved variant is the `loop` instruction, which has a similar syntax, but is only executed the first time the loop is entered. The counting register is automatically decremented when the end of the loop is reached, and a check for zero is performed. According to the manual, using the `loop` instruction results in a much better performance than using the `JNED` instruction. Although there is no information about nested loops, it is stated in [Inf03b, p. 26] that the loop cache only comprises two levels. Consequently, having more than one inner loop does not enhance the performance.

2.4.4 Timing Analysis[28]

The TriCore processor consists of two computational units; the integer processing (IP) and the load/store (LS) unit. The former is responsible for nearly all arithmetic instructions, whereas the latter is mainly responsible for address calculations and memory accesses. Disregarding conditional branches, pipeline hazards or multi-cycle instructions, all instructions take one clock cycle to finish. Moreover, it is also possible to execute one IP and one LS instruction – which do not depend on each other – in one clock cycle.

Pipeline stalls may occur when dealing with multiply accumulate instructions where one instruction uses the destination register of a preceding SIMD instruction as source register. In all other cases, the cycle duration is highly dependent on the type of instruction that follows. Infineon gives some recommendations for sophisticated instruction scheduling strategies. For example, a simple idea is to alternately use LS and IP instructions. The interested reader is referred to the manual and the guide for compiler writers for additional information.

2.4.5 Conclusion

Although the TriCore unites nearly all presented *special features* of the Blackfin, ARM and 32-bit AVR processors, analyzing the exact execution time of a given program is a quite challenging task. Firstly, one has to take into account the instruction scheduling; changing the order of only two instructions may cause a completely different result in the cycle count. Secondly, the static branch prediction model is very sophisticated and the exact number of cycles depends on the jump type (forward or backward jump).

Moreover, the possibilities when dealing with conditional instructions are very limited – in comparison with the ARM instruction set, as well as with the 32-bit AVR: Applying a single-path conversion to a given algorithm surely reduces the complexity of the CFG, but may have a tremendous effect on the WCET, which will certainly increase. Nevertheless, the Infineon TriCore processor provides some very interesting and useful instructions for WCET analysis, although the complex hardware implementation may cancel out the advantages of these features.

[28] Confer [Inf03a, p. 264-275] and [Inf03b, p. 30-43].

2.5 Tensilica Xtensa

2.5.1 General Features[29]

In contrast to all other presented processors of the current chapter, the Tensilica Xtensa is no off-the-shelf ASIC, but provides the developer a basic set of features which may be fully adapted to personal needs. This ranges from the individual scaling of the size of cache, RAM and ROM to the implementation of custom instructions. In contrast to other FPGA-based processors[30] like Altera's NIOS II, the Xtensa is delivered with a complete toolchain allowing the customer to adapt hardware, compiler, and software individually. Additionally, the processor may be optimized for different characteristics like small die area or low power consumption.

The main application field of the Xtensa processor lies in hardware-software codesign: When there is an application-specific problem to be solved, which requires high performance of a single task and the general-purpose instruction set of an average processor, the Xtensa processor may be an adequate choice. Section 2.5.4 gives some practical examples of how to increase the performance of an application with the help of additional instructions implemented in hardware. The following sections give an overview of the Xtensa architecture and instruction set. At the end, the predictability of the processor is evaluated.

2.5.2 Architecture Overview

As already mentioned, the Xtensa architecture is not fixed and may be configured and extended. In principle, it consists of five different types of modules/features:[31]

Base ISA Features These modules are the only fixed parts of the Xtensa. They implement the basic instruction set, basic processor control features, memory access, and the processing pipeline, which consists of five stages (see below).

Configurable Functions Like the base ISA features, these modules are part of every Xtensa implementation, but their parameters can be set to user-defined values. They include the register file, the instruction memory and the data memory. The memory size of each module may be chosen from a given list of options.

Optional Functions In contrast to the configurable functions, optional modules may or may not be added to the current processor design. They include a JTAG controller, a debugging interface and several ALU features like multiply-accumulate, DSP and floating point operations.

Optional and Configurable Functions These modules may or may not be added to the current processor design. They include data and instruction caches, data and instruction RAMs, timers, and an extension for exceptions and interrupts. Moreover, the user can specify several characteristics, such as cache size and cache replacement strategy.

[29] Confer [Gon00].

[30] Such processors are usually denoted *soft cores*.

[31] Confer [Ten02, p. 1ff] and [Ten10, p. 6].

Designer-Defined Features These modules provide the highest freedom for the designer. With the help of a description language, the user may add individual modules for the ALU, thus providing application-specific instructions. Moreover, it is also possible to implement a user-defined load/store unit or to adapt the pipeline to support SIMD instructions.

The Xtensa pipeline consists of five stages, which are similar to a typical RISC architecture. They include instruction fetch, instruction decode & register fetch, execute, memory access, and write back. Usually, an instruction takes five (pipeline) cycles to complete, but there are two exceptions: A memory load may cause a so-called *bubble cycle*, which is introduced by the next instruction if it depends on the result of the load operation. The second exception are *branch instructions*, which cause two instructions within the pipeline to be invalid if the branch is taken. Consequently, nearly all base instructions finish within one clock cycle, but load instructions may introduce one additional cycle and branch instructions two.[32]

2.5.3 Xtensa Basic and Optional Instruction Set[33]

The basic set of the Xtensa consists of 82 so-called *core instructions*, including load/store, move, basic arithmetic, shift, logical, branch, call and processor control instructions.[34] The basic instruction set also supports conditional moves allowing the application of a single-path conversion on a given code. Without any extensions, all instructions are encoded with 3 bytes, but the *code density module* allows 16-bit encoding of commonly used instructions. This can be compared to the Thumb instruction set of ARM processors as presented in Section 2.2.5, although the Xtensa does not need a special instruction for switching.

Another extension to the basic instruction set is the hardware-supported loop. The mechanism works similar to the hardware loops of the Blackfin and TriCore processors, but nested hardware loops are not supported. Nevertheless, there are fewer restrictions, meaning, for example, that calls and conditional branches within the loop boundaries are allowed. Nevertheless, the programmer has to take care that the last instruction of a loop is no branch or call, and there are also some limitations on the start and end addresses of the loop.

There are several extensions for arithmetic functions including 16- and 32-bit multiplication, 16-bit multiply-accumulate, 32-bit integer division, and minimum and maximum calculation. Nearly each of them may be implemented separately. Moreover, a floating point coprocessor may be implemented, providing IEEE 754 single-precision floating point operations. Another interesting instruction set extension is the multiprocessor support providing synchronizing functions for shared memories. Of course, there are several other interesting extension modules, e.g., for cache management, which will not be discussed here. The interested reader is referred to the Xtensa ISA manual [Ten10], where all features are described in detail.

[32] Confer [Ten02, p. 67f].
[33] Confer [Ten10, p. 71-105]
[34] See Table 3-11 in [Ten10, p. 33].

2.5.4 Individual Instructions[35]

One of the most interesting features of the Xtensa processor architecture is the possibility to implement fully customized instructions with the so-called Tensilica Instruction Extension (TIE) language. The programmer may define the opcode, the assembler name, the used data types (e.g., integer), the semantics, and some other characteristics of the new instruction. The delivered assembler and C compilers automatically recognize the newly defined instructions, so that no additional compiler adaption is needed. Moreover, the Xtensa design framework automatically tries to adapt the new instructions to the existing processor settings, i.e., by adding the corresponding features to all relevant pipeline stages. Additionally, the framework automatically tries to verify the new instructions, if test vectors have been provided.

Gonzalez gives an impressive example of how a hardware-accelerated DES implementation may cause a speed-up of more than 50 times in comparison to a software-based solution, while only needing 4.500 additional gates of hardware. Some other examples include JPEG image compression, FIR filtering, and motion estimation for video streaming.[36] Consequently, the Xtensa seems a quite feasible solution for applications which benefit from specific hardware-implemented operations while still offering all relevant features of a general-purpose processor.

2.5.5 Conclusion

The Xtensa processor cannot easily be evaluated from a real-time system designer's point of view because there are so many different features which can be part of the processor and could make WCET analysis difficult. Nevertheless, as the Xtensa allows to design a processor without caches and complex SIMDs or floating point instructions, but at the same time provides hardware support for conditional moves and loops, it may be an interesting choice for hard real-time systems; the low performance due to missing data and instruction caches can be strongly improved by implementing a user-defined function perfectly fitting the needs of the application. Unfortunately, it is not possible to change some fundamental internals of the processor, like predicated instructions or support for nested hardware loops. Consequently, the Xtensa processor is the ideal choice for systems with individual needs which cannot be fully met by general-purpose processors or ASICs.

2.6 Conclusion

All processors presented in the current section have advantages and disadvantages with respect to WCET analysis. Table 2.1 gives an overview of the most significant features of each processor. Although the 8-bit AVR shows good characteristics for timing analysis, it does not provide any support for single-path conversion. Moreover, its limited instruction set and low performance on complex calculations do not make it very attractive when dealing with sophisticated real-time tasks. Only ARM processors support a fully predicated instruction set, which makes them very

[35] Confer [Ten02, p. 59-62] and [Gon00].

[36] Some of the named examples were presented in [Gon00], but experienced an additional speed-up, as may be verified in [Ten02, p. 3].

Processor / Microcontroller	Register Bit Width	Instruction Size (Bytes)	SIMD	Support for Single-Path Conversion	Cache	Timing Analysis
Blackfin	32	2 - 4[a]	yes	−	yes	difficult
ARM	32	2 - 4	yes	+	yes	difficult
8-bit AVR	8	2 - 4	no	−	no	easy
32-bit AVR	32	2 - 4	yes	±	yes	difficult
TriCore	32	2 - 4	yes	±	yes	difficult
Xtensa	32	2 - 3	yes[b]	±	yes/no[b]	medium[b]

[a] May be 8 bytes for a parallel instruction.
[b] Depending on the concrete implementation.
[c] May be up to 8 bytes for extended instruction set.

Table 2.1: Feature overview of all presented architectures of the current section.

feasible for single-path conversion. Nevertheless, the timing analysis of a given code is a quite challenging task due to the elaborate pipeline model and the involvement of caches.

Only ARM, 32-bit AVR processors and the TriCore processor may be used for single-path conversion. Unfortunately, the TriCore has an even more sophisticated internal architecture than ARM processors and provides only limited support for conditional execution of instructions. When directly comparing ARM and Atmel processors, ARM provides better support for single-path conversion, whereas Atmel offers better features for performance measurements. Finally, the Blackfin processor may at the first glance be the totally wrong choice when implementing a real-time system, but it provides some convenient features like hardware loops and cache memory management, which could make it quite attractive when operating in soft real-time environments.

CHAPTER 3

Time Predictable Architectures

3.1 The Necessity of Time Predictable Processors

As could be seen in Chapter 2, the design of modern (embedded) processors is mainly based on improving the performance of average case scenarios and providing functions for multiple data processing (SIMD). This trend can be identified over the past decades, although there have been proposals for predictable processor architectures for about 20 years. The main problem is, that on the one hand, the average case performance of processors may be successfully improved while on the other hand, the worst-case execution time might not be calculated any more due to complex constraints like cache and pipeline status or even different types of memory access strategies. Moreover, improvements for the average case may result in worse performance for worst-case scenarios.

As processor design gets more and more sophisticated while at the same time the complexity of hard real-time and highly dependable systems increases, the necessity of simple and predictable architectures arises. For example, many current state-of-the-art embedded applications make use of multi-threading. It is true for most contemporary processor architectures, that the timing behavior of each task depends on the execution and status of the other tasks, thus making reliable WCET analysis nearly impossible. Martin Schöberl outlines five points which will be of major concern for highly predictable architectures of future real-time systems (see [Sch09b]):

(i) There is a mismatch between performance-oriented computer architectures and worst-case analyzability.
(ii) Complex features result in increasingly complex models.
(iii) Caches, a very important feature for high performance, need new organization.
(iv) Thread level parallelism is natural in embedded systems. Exploration of this parallelism with simple chip multiprocessors is a valuable option.
(v) One thread per processor obviates the classic schedulability analysis and introduces scheduling of memory access.

Thiele and Wilhelm give a definition of *performance* and *predictability* in the context of hard real-time systems (confer [TW04]): According to them, the performance of a processor may be defined as the reciprocal value of the execution time. This means, that processors with low execution times have high performance and processors with high execution times have low performance. There has to be distinguished between best, average and worst-case performance because a good average execution time does not necessarily imply good timing results for the worst case and the other way round. Predictability may also be defined for lower and upper bounds, i.e., predictability of best and worst case: some processors may behave quite predictably for best case scenarios, but are too complex for providing predictability for worst-case scenarios. Consequently, the overall predictability of a processor or system architecture also depends on the predictability of all relevant scenarios.

In 1997, when WCET analysis was still mainly based on manual calculations and measurements, Zhang (confer [Zha97]) tried to figure out techniques to improve predictability. He was already aware of the complexity which comes along when introducing caches, especially if the analysis includes preempted or interrupt driven real-time systems. Therefore, he suggests that the cache should be partitioned into several segments such that every task has an independent cache set. Many presented architectures of the current chapter use a similar approach. To improve predictability of processor pipelines, he refers to the shared pipeline approach which was originally implemented by Cogswell et al. in the MACS architecture (see Section 3.3 for further details). As Zhang tried to involve many aspects of real-time systems, he also enumerates several factors influencing the predictability which can usually not be influenced by the hardware design of a processor. These include unpredictability caused by sampling analogue signals, communication protocols, clock drifts and context switches initiated by the operating system. Although they may all be the reason for uncertain timings, they will not be taken into account in the current thesis.

Heckmann et al. presented one of the first tools for static WCET analysis in 2003 (confer [HLTW03]). They implemented analyzing algorithms for architectures with different levels of complexity, including various cache replacement strategies and pipeline architectures. Although the results seem quite convincing, the authors admit that all implemented architectures were analyzed and implemented manually. This means that it takes an enormous effort to extend the tool for other processors because every architecture has unique characteristics. In the end, they give some advice for designing predictable processors or processors which can be analyzed statically. This guidance includes to make use of simple cache architectures, i.e., separate caches for data and instructions and make use of simple cache replacement strategies like LRU (least recently used). Moreover, the processor should only rely on static branch prediction and should not support out-of-order execution. Last, the design should avoid *short-cuts* causing a speed-up for certain scenarios which may only be rarely relevant. The problem of static WCET analysis is not related to a single architectural feature, but to the side effects which multiple features may have on each other (e.g., dynamic branch prediction and caches).

The current chapter tries to give an overview on different approaches of implementations of time predictable architectures in the past twenty years . They cover general design approaches for whole real-time systems like the Spring Architecture presented in Section 3.2 as well as very detailed proposals for time predictable processors. Some of them also have been implemented

in hardware (e.g., on an FPGA) such that concrete performance statistics exist. Nevertheless, in most cases, the architectural specification spares a description of the implemented instruction set. Hence, there do not exist any general guidelines, stating which instructions should be part of a predictable processor.

3.2 The Spring Architecture

In [Sta90], John Stankovic presents the so-called *Spring Architecture*, which is a general design approach for real-time systems. It includes the separation of the operating system, which is executed on a *system processor*, from the user tasks, being executed on one or more *application processors*. In contrast to the approach to provide a critical and a non-critical CPU, this architecture has the advantage that the current status of the operating system may not influence the execution of a critical task. In case of a critical and non-critical CPU, the application and the operating system are executed on both processors and may influence each other. In addition to the CPU segmentation, both processors have their own I/O areas.

The Spring Architecture also introduces partitioning for resources (e.g., sensors or actuators) and for tasks. This means that tasks may be of three different types (critical, essential and non-essential) and that the needed resources and worst-case behavior is known a priori. Consequently, the Spring kernel, which is implemented on all nodes of the distributed real-time system, can decide at run-time which task shall be scheduled next. Nevertheless, critical tasks have to be scheduled a priori. Such, it is possible to provide *graceful degradation* – all critical tasks meet their deadlines, but not all essential tasks may finish on time in a worst-case scenario.

Stankovic outlines the necessity for using *simple* processors, i.e., RISC processors, which accomplish executing one instruction in a constant number of CPU cycles at any time. The predictability of the used processors is important for the static WCET analysis of the critical tasks. One problem which may arise when making use of co-processors, e.g., for floating-point calculations, is that such instructions may involve uncertainty when interrupts or exceptions occur. Stankovic therefore suggests that co-processors should only be used if they are not pipelined and are only capable of executing a single operation at once. Other external resources which may be needed are memory management units, DSPs and DMA (direct memory access) controllers. They have to be integrated into the system such that they may not have any influence on time-critical tasks.

To summarize, the Spring Architecture may be seen as an interesting design proposal for a real-time system as a whole, but does not present a suitable solution for real-time processors. Although highly predictable processors are a requirement for reliable hard real-time systems, they are simply assumed as existing: Stankovic only states that an ideal real-time processor does not include caches or pipelines, but still has to show good performance – an assumption which might not be fulfilled in most cases. Moreover, the introduction of sophisticated dynamic scheduling algorithms may cause the system to become indeterministic and therefore static system analysis is out of discussion. Nevertheless, Stankovic's call for using simple components which provide high predictability, is still up to date and can be seen as one of the key aspects when designing modern real-time systems.

3.3 MACS

The MACS (Multiple Active Context System) architecture was introduced in [CS91] by Cogswell and Segall and describes a predictable multitasking processor. The processor uses a pipeline, but does not have any caches. To provide a predictable pipeline and constant cycle counts for each instruction, the authors implemented a *shared pipeline* for multiple tasks, which are called *contexts*. This means that a context switch is automatically executed every clock cycle. All tasks are executed periodically in a round-robin manner. So, when there are N tasks supported by hardware, each context may execute a single instruction every N^{th} clock cycle. The MACS architecture denotes the period, until all tasks have made a complete execution step, a *major cycle*, which takes N clock cycles. This kind of shared pipeline is based on two ideas:

(1) Usually, there are no dependencies between any two instructions of two different tasks, except for memory accesses to shared memories, which will be discussed later.
(2) When supporting a feasible number of contexts, all instructions may finish within a major cycle such that there do not exist any dependencies between two following instructions of a single task. This means, no branch prediction is needed to speed up conditional jumps and the pipeline does not need to support register forwarding or similar techniques.

Although this approach seems highly inefficient at the first glance, the authors argue that the average performance is comparable with a cached processor while still remaining predictable: The average cycle count per instruction (CPI) is calculated by

$$\text{CPI}_{\text{cached}} = r_{hit} \cdot c_{hit} + r_{miss} \cdot c_{miss} = r_{hit} \cdot c_{hit} + (1 - r_{hit}) \cdot c_{miss} \tag{3.1}$$

in the case of a cached processor. r_{hit} indicates the statistical cache hit rate, r_{miss} the miss rate and c_{hit} and c_{miss} the cycle count in case of a cache hit or miss, respectively. Given the MACS architecture, one major cycle consists of N CPU cycles. N is determined by the amount of tasks/contexts supported by the processor. Assuming that only U out of N contexts are really used and that all instructions need one major cycle to finish, the average cycles per instructions are defined as

$$\text{CPI}_{\text{MACS}} = \frac{N}{U}. \tag{3.2}$$

Assuming, one wants to implement a MACS architecture having the same average performance as a cached processor. Let the clock frequency of both processors be equal. If we want to find out the smallest number of tasks, such that MACS provides the same CPI as the cached architecture, we have to set (3.2) equal to (3.1):

$$\frac{N}{U} = r_{hit} \cdot c_{hit} + (1 - r_{hit}) \cdot c_{miss} \tag{3.3}$$

From this formula, it follows that

$$U \geq \left\lceil \frac{N}{r_{hit} \cdot c_{hit} + (1 - r_{hit}) \cdot c_{miss}} \right\rceil. \tag{3.4}$$

Let us now assume that a worst-case memory access takes M CPU cycles. Consequently, we have to choose the number of supported tasks by MACS to be at least equal to M such that all

instructions may finish within one major cycle. Moreover, let us assume that a memory access to cached data ideally takes one clock cycle. Applying this to (3.4), we can calculate the minimum number of used tasks U for a MACS architecture which provides the same CPI as the cached processor:

$$U \geq \left\lceil \frac{M}{r_{hit} + M \cdot (1 - r_{hit})} \right\rceil \tag{3.5}$$

Setting M to 16 and r_{hit} to 90 % reveals that the number of used tasks has to be at least 7 – less than 50 % of the available contexts. In a scenario of 95 % hit rate and a worst-case memory access time of 8 cycles, the number of used tasks has to be at least 6. Even for this scenario, the processor usage is only about 75 %.

Although the MACS architecture provides considerable performance in comparison to cached processors, one open point is the memory organization: The programming model which is implemented by each context is the so-called *uniform memory access* model. This means that every task has its own local *memory bank*, which has a guaranteed access time of one major cycle. Consequently, access to local memory banks has no influence on the timings of other tasks. Moreover, every context has access to shared global data. The access time to global data is dependent on the amount of contexts concurrently working on the same data set. The more tasks want to access this shared memory at the same instant, the longer a task might have to wait. In the worst-case scenario, all N contexts of MACS are used and all of them want to access the same shared memory location. It may now take up to N major cycles until all tasks may have completed the access. Nevertheless, Cogswell and Segall argue that the remaining unpredictability, i.e., the scaling factor between best and worst-case execution time, is still much less than for a typical cached processor. This statement is also confirmed by simulation results, which show a maximum divergence factor of 2 in case of a MACS architecture supporting up to 16 contexts whereas a typical cached architecture may have a factor of 2.5 or more. Still, the question arises whether a profound static WCET analysis delivering an exact cycle count is still feasible for the MACS architecture or whether it is only possible to provide a reduced jitter for soft real-time applications.

Summarizing, the MACS architecture is an interesting approach towards a predictable processor and was one of the first evaluated architectures in this research area. Unfortunately, the evaluation is only based on simulation and there does not exist a hardware implementation of MACS. For the simulation, the instruction set of MIPS R2000 has been used. It was extended by special instructions for hardware supported task management (e.g., fork, lock, etc.). Beside the fact that MACS has never been realized in hardware, another aspect has to be considered: The discrepancy between memory and processor speed has still increased over the past 20 years, resulting in even higher clock cycle delays for memory accesses than in 1991. Consequently, a current implementation of MACS may require a relatively high number of supported contexts to guarantee access to memory within one major cycle. This problem could be avoided by implementing a multi-processor or multi-core realization of MACS, which has also been suggested by Cogswell and Segall. Unfortunately, there has not been any further research about the MACS architecture, but modern proposals for predictable processors like the PRET architecture presented in Section 3.8 refer to similar concepts. This indicates that MACS might have been a first step in the right direction.

3.4 SPEAR

The **S**calable **P**rocessor for **E**mbedded **A**pplications in **R**eal-time Environments (SPEAR) was developed by the Institut für Technische Informatik of the Vienna Univeristy of Technology, confer [Del02, DHPS03]. According to the authors, there are two main requirements for processors working in real-time environments:

(1) The response time of the processor shall be as small as possible. It is of utmost importance that time-critical tasks meet their deadlines. A general purpose processor might not be able to handle all tasks correctly in a worst-case scenario due to its indeterminism.

(2) The response-time jitter, i.e., the variability of the response time, shall be as small as possible. An ideal predictable real-time processor has no jitter at all. Consequently, a task always reacts to certain events after a constant time, disregarding the internal and external status of the real-time system.

As most real-time systems have to interact with the external environment, they have to be able to react to *asynchronous* events within a minimum period of time. This can often only be guaranteed by using interrupts. Unfortunately, interrupts usually may occur at any instant such that the required context switch might have variable response times, depending on the internal state of the processor. To guarantee a predictable behavior in all possible scenarios, Delvai et al. propose the following implementation:

Synchronization: In order to react predictably to asynchronous external events, it is necessary to synchronize all involved operations. First, the signal causing an interrupt has to be synchronized, i.e., the digital representation of the signal has to be present at the beginning of the next clock cycle. Afterwards, the instruction synchronization is responsible that an interrupt service routine may only be executed at safely defined points in time. This synchronization may only be fully implemented if the processor is predictable and does not include sophisticated features like out-of-order execution, multi-cycle instructions, etc.

Context Switch: The context switch consists of two phases, namely saving the current context and identifying the corresponding service routine. Although hardware supported context switches are more efficient, they introduce unpredictability and may increase the response-time jitter. That is the reason why the authors of SPEAR decided to use the traditional software stack-based solution. Identifying the corresponding interrupt routine to an external event may become difficult when there are multiple possible sources. Therefore, it is required that the mapping of an external signal is deterministic and is guaranteed to finish on time.

Interrupt Service Routine: Usually, the major part of the response time of a real-time system depends on the implementation of the interrupt service routine. Although it is not in scope of influence of the processor to provide predictable algorithms, it can support the programmer with several instructions for single path conversion etc.

To accomplish the just described requirements, the SPEAR architecture shows several features: It implements a so-called 16-bit core, meaning that all registers, the data and address bus as well as the instructions have the same fixed size of 16 bit. Thus, instruction fetch, decode and data access are quite simple and do not require sophisticated mechanisms. To achieve an acceptable average performance, the SPEAR is a pipelined architecture, but avoids data and control hazards by operand forwarding and pipeline clearing. Such, it is possible that every instruction is executed in a constant number of CPU cycles, regardless of the current processor state. To avoid a memory bottle neck, it implements a Harvard architecture and has separate instruction and data memories of 4 KByte each.

The instruction set of the SPEAR comprises 80 instructions, including operations which are only executed if a condition flag is set. The concept is similar to the ARM predicated instructions, which are described in Section 2.2.4. Unfortunately, some of the most common instructions like add or subtract do no support conditional execution meaning that the single-path conversion of some algorithms may cause code bloating.

Except for the just described conditional instructions, the SPEAR instruction set does not provide any special operations like SIMD or DSP instructions. Moreover, SPEAR does not support floating point operations or integer division and multiplication. That is the reason why the authors give the designer the possibility to add so-called *extension modules*. In contrast to the Xtensa processor by Tensilica – see Section 2.5 for further details – these modules are not integrated into the internal design, but may be compared to coprocessors. The SPEAR provides memory dedicated registers for data exchange between the processor core and the extension modules. The modules may either be individually implemented in a hardware description language or chosen among existing ones like, e.g., a UART.

The SPEAR architecture is a very interesting approach towards a time predictable processor and unites a neat and easy to analyze instruction set with the flexibility introduced by the extension modules. Moreover, in contrast to other architectural proposals of the current chapter, it has been implemented and evaluated on real hardware. Nevertheless, real-time systems involving complex tasks like DSP or floating point operations require sophisticated hardware implementations of extension modules to guarantee predictability and constant response times for worst-case scenarios.

3.5 VISA

The Virtual Simple Architecture (VISA) was presented by a research group of the North Carolina State University in [ASP+03]. It is based on the following idea: Although complex processors are not feasible for WCET analysis, they show better average performance and may also have less execution time for worst-case scenarios. Simple architectures may be easily analyzed but usually only provide low performance. The VISA approach suggests to use an architecture which is capable to dynamically switch between a complex and a simple processor implementation. The complex pipeline may realize dynamic branch prediction and caches, which can be disabled on demand. Consequently, if the complex architecture is not able to meet deadlines which have been calculated for the simple architecture, a switch to the simple architecture is performed.

In order to guarantee that a switch does not cause a deadline to be missed, the deadline calculation takes the needed overhead into account. Moreover, a task is divided into multiple sub-tasks. Whenever a sub-task finishes, a watchdog application checks whether the execution of the whole task is still on time. If that is not the case, the switch to the simpler architecture is performed, otherwise, the next sub-task is executed.

The instruction set of VISA is based on the so-called Simplescalar instruction set, which has been developed by the University of Wisconsin. The authors argue that like ISA, VISA shall be an abstraction from the really used architecture and therefore may be ported to different processors. Although this seems to be a reasonable approach, VISA has only be evaluated by a cycle accurate simulator and there is currently no hardware implementation. Thus, no statement can be done whether VISA is flexible enough to be used for different processors or instruction sets.

The simulated benchmarks however show that the VISA approach is quite applicable for real-time systems in low power environments: In all tested scenarios, the complex architecture shows better worst-case behavior than the simple variant. Consequently, either the processor may be switched into a less energy consuming sleep-mode when a task has finished, or the supply voltage may be reduced, resulting in smaller clock frequencies and less average energy consumption. Anantaramen et al. showed that the VISA approach may yield energy savings of 40 to 60 % in comparison to a conventional architecture.

Concluding, the VISA approach shows good average performance while satisfying hard deadlines and reducing energy consumption. Nevertheless, it is no applicable solution in environments requiring low jitter. Moreover, dividing a time-critical task into multiple sub-tasks and setting up safe deadlines requires sophisticated techniques, which have to be adapted for different applications. Finally, a hardware implementation of the presented approach is necessary to evaluate the performance in comparison to other existing time predictable architectures.

3.6 JOP

The Java optimized processor, abbreviated JOP, was developed and implemented by Martin Schöberl during his PhD thesis. The processor is also in industrial use and was improved and extended by several users and students of technical universities in Vienna and Copenhagen. Beside from Schöberl's PhD thesis [Sch05b], there are several conference papers presenting different aspects of the processor, e.g., performance and real-time garbage collection, and a manual [Sch09a], which can be seen as an updated version of the PhD thesis. The whole JOP project, including the programming tool chain and VHDL code, is available in open source under the GNU General Public License, version 3; see [Sch11] for more information.

Although the main intention of JOP was to provide an implementation of the Java virtual machine (JVM) in hardware, the processor should also be applicable for real-time systems. Consequently, the architecture tries to support all features of a software implementation of the virtual machine while remaining predictable and providing support for WCET analysis. Like the JVM, JOP is a stack-based architecture, but implements a reduced instruction set. This design decision is based on two observations:

(1) A reduced instruction set allows simple encoding of instructions in opcode. JOP only uses 8 bits for each of its 43 instructions.[1] Additionaly, a fixed bit width for the microcode makes instruction fetching and decoding easier and is thus more predictable.

(2) The JVM provides some instructions which are usually implemented by system calls or software libraries and cannot easily be implemented in hardware. Such complex instructions are emulated by multiple simple operations which are supported by JOP.

The pipeline of JOP has four stages: The first is responsible for fetching Java bytecode, which is the usual output of a Java compiler. All existing JVM instructions are saved in a ROM table in the second stage. The first pipeline stage is also responsible to map a Java bytecode to the corresponding ROM address. A JVM instruction in the ROM table may be implemented in two different ways:

(1) The Java bytecode is directly mapped to one JOP instruction. Consequently, the second stage generates a signal, indicating that the first pipeline stage may fetch the next Java instruction.
(2) The Java bytecode has to be emulated by multiple JOP instructions. In this case, the ROM address corresponds to the starting address of a sequence of JOP instructions. This principle may be compared to a function call. The last JOP instruction is responsible to generate a *next* signal which triggers the first pipeline stage to fetch the next JVM instruction.

Note that the described approach requires two different program counters: one Java program counter indicating the next address of the Java bytecode and one JOP program counter for accessing the next JOP instruction. The third pipeline stage is responsible for the decoding of JOP instructions and address generation. The last stage executes an operation and saves the result in the designated destination register. As JOP implements a stack-based architecture, the result is always available at the same register and there is no need for a separate write-back stage.

JOP has about 40 instructions, which are able to implement the complete functionality of the JVM. All JOP instructions are executed within one cycle, except for the *wait* instruction. The latter may be used to implement pipeline stalls for memory accessing instructions. The JVM of JOP consists of three pipeline stages because the Java bytecode fetching stage can be disregarded. Thus, two consecutive wait instructions are necessary to fill the (JVM) pipeline. The execution time of the wait instruction depends on the timing of memory accesses. Hence, the usage of predictable memories is necessary to support WCET analysis.

Although JOP was mainly designed to be predictable, it should also show a good average performance. Thus, JOP makes use of separate data and instruction caches. The data cache is called *stack cache* because the whole memory is organized in a stack-based manner. It does not need to be very complex, because only top elements of the current stack may be accessed. For the instruction cache, Schöberl introduced a so-called *method cache*. It is based on the following idea: Java bytecode mainly consists of small basic blocks and many function and method calls. In order to remain predictable, a basic block in the instruction cache is only

[1] In fact, there is a *nxt* and *opd* bit for each instruction such that the effective length is 10 bits.

replaced when a method call occurs. The cache replacement strategy is a simple FIFO (first in first out) implementation and does not consider whether a memory region has indeed been accessed. In comparison to a direct-mapped cache, the presented approach is not significantly slower, but is predictable and is therefore applicable for static WCET analysis.

In [Sch05a], multiple implementations of the JVM including JOP are compared concerning their performance executing relatively small Java applications. The limitation to short programs is due to the typical application area of JOP, which lies in embedded real-time systems. This is the reason why the software based JVM running on an Intel MMX processor shows the best performance: The whole program fits into the cache of the processor. The evaluation may also be seen somehow problematic because the presented solutions are based on completely different architectures with several clock rates, memory and cache sizes and other factors. To remain somehow comparable, the performance is also evaluated relatively to the clock frequency of the corresponding processors. In comparison with 8 other processors, JOP shows the highest absolute and relative performance after the already mentioned software based solution executed on an Intel MMX processor. Moreover, JOP based implementation shows the smallest difference between best and worst-case execution time resulting in low jitter.

Like the SPEAR, the Java Optimized Processor has been implemented in hardware and is also in industrial use. In contrast to SPEAR, it implements the existing instruction set of the JVM by partially emulating the most complex instructions. Moreover, SPEAR is a classic load-store architecture whereas JOP is stack-based. By introducing a predictable cache architecture, average and worst-case performance is increased. WCET analysis may be performed on different levels, such as Java bytecode or the resulting JOP instructions. Although Java was originally designed for user applications, the development of real-time Java made it an interesting choice for dependable system developers over the last years. As could be shown by Martin Schöberl, it is possible to implement a Java based processor which exhibits competitive performance and is still predictable. Consequently, JOP may be a valuable choice for future applications in the segment of real-time systems.

3.7 MCGREP

The MCGREP (microprogrammed coarse grained reconfigurable processor), presented by Jack Whitham and Neil Audsley in [WA06], follows a completely different approach to the so far presented solutions of the current chapter. Their design is based on the idea that application specific processors provide better performance than general purpose processors when they are restricted to certain tasks. Consequently, implementing application specific instructions may dramatically improve the execution time while the processor is still predictable. Although this approach appears to be quite similar to the Tansilica Xtensa processor, presented in Section 2.5, Whitham and Audsley go one step further: MCGREP provides instructions to reconfigure the given hardware implementation during runtime.

Witham and Audsley differ between simple CPU architectures such as the Motorola 68000, complex architectures like the Intel Pentium, application specific instruction set processors like the Xtensa and reconfigurable architectures, which are mainly implemented on FPGAs. These reconfigurable processors allow dynamically changing the instruction set of the processor, re-

sulting in higher flexibility. According to the authors, a perfect processor for real-time systems provides high throughput, high predictability, high flexibility and low transistor count. A low transistor count results in a small die area, which corresponds to less power consumption and small prices. High flexibility is needed if the same processor shall be used for various specific tasks.

Although a reconfigurable architecture usually has a high transistor count and is not predictable, Witham and Audsley showed that MCGREP may be implemented with medium hardware effort and is still applicable for real-time systems. A first version of the MCGREP architecture implements a simple two stages pipeline, namely instruction fetch & decode and execute & write back. Every instruction may either be executed within a single clock cycle or stalls the pipeline until it is finished. Nevertheless, the execution time is constant and independent from the current pipeline status or other instructions. The implemented instruction set is based on the open source soft core OpenRISC, which has been slightly adapted to support instructions for reconfiguration. In this basic configuration, the MCGREP shows similar performance to MicroBlaze, another soft core processor, and the OpenRISC processor, but has slightly less throughput in some benchmarks.

To increase performance while remaining predictable, MCGREP does not implement a cache, but allows the reconfiguration of the current hardware within one clock cycle. This means that dedicated multiplexers and functional units may be activated or disabled by special microcode instructions.[2] This technique allows to provide hardware support for hot spots and worst-case paths within a program, which have to be identified in advance. Unfortunately, this process had to be done manually and is an issue for future improvement.

Evaluation shows that the throughput of MCGREP nearly remains constant when a task gets periodically interrupted by another task, whereas the execution time of MicroBlaze and OpenRISC highly depends on their instruction cache. In more than 70 % of the presented benchmarks, MCGREP even shows a higher worst-case performance than MicroBlaze. Moreover, the microcode of MCGREP may be used to provide real-time operating system support such as task and interrupt priorities, context switching and atomic actions. Consequently, the predictability of an operating system only depends on its implementation and not on the underlying hardware architecture.

In [WA08], Whitham and Audsley added a so-called *trace scratchpad* to the existing architecture. Traces include multiple micro operations, which are executed in a given order or at the same time. They were first introduced to simplify the generation of VLIW code. Consequently, the compiler has to identify parts of given code and generates traces, which help to optimize the worst-case path. The statically generated traces may be loaded to the scratchpad on given points in the program to preserve predictability. In the case of conditional branches, it may happen that another trace which has not been loaded to the scratchpad becomes valid. Now, the processor has to switch back to normal execution and the micro operations in the scratchpad become invalid. Thus, new worst-case scenarios may arise, which all have to be taken into account when traces for scratchpads are generated.

The presented approach by Whitham and Audsley is based on a completely different idea than the other architectures of the current chapter. Nevertheless, MCGREP shows medium av-

[2] Normal MCGREP instructions are denoted *machine code*, instructions to reconfigure the hardware *microcode*.

erage performance and provides flexibility and high predictability. Unfortunately, the presented toolchain involving analyzing hot spots and worst-case paths in programs and manually generating microcode is not very feasible for real life applications.

3.8 PRET

The Precision Timed Machine was firstly presented by Stephen Edwards and Edward Lee in 2007 in [EL07]. They show that current processors are not feasible to provide reliable or predictable execution times because they are designed for improving average case performance, ignoring possible degradation in worst-case scenarios. Moreover, traditional layers of abstractions, like ISA or high-level programming languages, are no adequate solutions in case of real-time systems: Current instruction sets usually cannot guarantee a constant execution time for all operations because the exact duration commonly depends on the current pipeline and cache stage. Consequently, real-time operating systems, which need exact timings for each task, cannot give any guarantee that the system always meets its deadlines. Edwards and Lee therefore want to introduce PRET machines which allow static analysis and exact timing behavior. A first version of PRET was presented in [LLK+08].

Like the MACS architecture (see Section 3.3), the PRET architecture makes use of a pipeline shared by multiple threads. Consequently, stalling and clearing of the pipeline can be avoided as there are no dependencies between any instructions of different tasks. Moreover, except for memory accessing functions and the so-called *deadline* operation, all instructions may finish within one pipeline cycle, i.e., six CPU cycles. The pipeline consists of six stages and supports the so-called *replay* of an instruction if it takes more than six CPU cycles. Thus, the execution time of one task does not have any influence on other tasks.

The previously mentioned deadline instruction was first presented by Ip and Edwards in [IE06]. It is meant to provide a cycle accurate delay of a following instruction without using NOPs. In principle, the deadline instruction sets an internal countdown timer to a given value. The next instructions starts the timer such that the subsequently following instruction will be delayed until the timer reaches zero. Consider an easy example: a conditional branch usually takes different cycle counts, depending on the implemented branch prediction and the pipeline depth. With a preceding deadline function, the instruction following the conditional branch will not be executed until the deadline is over. Currently, nothing happens if the deadline is missed, but future extensions may cause an exception to be raised. Consequently, the deadline instruction may provide cycle accurate timing.

Like the MACS architecture, each task has its own local memory. Unfortunately, a memory access takes 13 clock cycles, which is far more than a complete pipeline cycle. Moreover, DRAM needs to be refreshed such that no data may get lost. Usually, a *burst refresh* is performed, meaning that the whole memory bank is not available for reading and writing. This strategy introduces unpredictability and is no adequate solution for the timing analysis of instructions involving memory. Liu, Reineke and Lee describe their solution in [LRL10]: A so-called *memory wheel* is responsible that a task may only access its dedicated memory bank within a specified period of time. Meanwhile, the memory banks of other tasks can be refreshed. The memory wheel grants access for 13 cycles to each task in a round-robin fashion and repeats its

schedule every 78 cycles ($13 \cdot 6 = 78$). In a worst-case scenario, a task misses its window by one cycle and has to wait for 77 cycles. Thus, a memory access may take up to 90 cycles.

In [LLK+08], a simple producer-consumer example is given, which makes extensive use of the deadline function for task synchronization. The consumer task starts a deadline counter such that it is guaranteed that the producer has at least written its first byte to a globally accessible array. Another presented application using the deadline instruction is a VGA (video graphics array) driver, which needs exact timings to generate a valid output signal.

As PRET is based on the instruction set of SPARC, the authors could easily evaluate the performance in comparison to the LEON3 processor, which is an industrial implementation of SPARC. Although the tested benchmarks did not include multiple tasks, PRET was only about 3 to 4 times slower than the LEON3. In a scenario with six concurring tasks, PRET may show even better performance than LEON3, but additionally provides predictability. Moreover, the benchmarks did not take the deadline instruction into account, meaning that they do not fully exploit all of the hardware features of PRET.

As already mentioned, the PRET approach uses similar techniques like the 20 year old MACS architecture. Nevertheless, modern hardware features like DRAM and scratchpads are used to improve performance and provide predictability. If the deadline instruction will be implemented such that a deadline miss triggers an exception, system designers may get a convenient means for testing and error detection. Moreover, static WCET analysis might become easier because the precise cycle count at specified points of a program are known. Although this approach seems to be similar to the watchdog points in VISA (see Section 3.5), the resulting cycle counts are constant and do not represent upper bounds. Future research will show whether PRET is indeed competitive to other industrial and academic solutions in the field of predictable processors.

3.9 Time-Predictable VLIW Processors

Although most modern processors are not based on a VLIW architecture, this approach has some remarkable advantages concerning predictability: Superscalar processors provide multiple functional units and out-of-order execution. Hence, they cannot guarantee reliable timings due to data and structural dependencies. The idea behind VLIW processors is to give the compiler the responsibility to set up a correct and possibly fast instruction schedule. One first step to adapt VLIW processors for real-time systems is to change the scheduling strategy of the compiler such that the worst case is taken into account.

Yan and Zhang tried to make an existing VLIW processor more predictable by redefining the instruction scheduler in [YZ08]. They implemented a simulator based on the HPL-PD architecture, which also provides predicated instructions. Their first approach did not take any caches into account to verify the correctness and evaluate the performance of the proposed algorithms. On the one hand, they implemented a *full-if* conversion function which replaces all basic if-then-else blocks with predicated pendants. This approach is indeed very efficient in the case of predicated VLIW processors because both branches can be concurrently executed without needing to perform conditional jumps. On the other hand, they implemented a so-called *intra-block nop insertion* to provide better predictability. This algorithm looks for joining branches

within the CFG of a program. It calculates the WCET of the last instruction of each branch and inserts the corresponding number of NOPs such that the first instruction of the joint path may be executed immediately. Consider the following example: There is a multiplication instruction which usually takes 3 cycles and an unconditional jump which takes 1 cycle to finish. The target address of the jump is the next instruction after the multiplication. Thus, depending on the current execution path, this instruction may have to wait until the multiplication has finished or can be executed immediately. To avoid this problem, the algorithm inserts two NOPs after the multiplication such that the next instruction always takes the same cycle count. Note that this technique usually does not change the resulting execution time, but eliminates complex side effects, which had to be taken into account for WCET analysis otherwise.

Although the primary goal of Yan and Zhang was to provide a reliable method to increase the predictability of VLIW processors, the evaluation of their benchmarks showed that they also lowered the WCET in all but one test run. In a second step, they also considered caches and calculated the WCET based on static simulation. Note that the usual complexity of WCET analysis including caches is mainly based on the fact that branch prediction, pipeline and cache state are highly coupled. As the pipeline of VLIW processors is statically determined, the calculated WCETs were quite conform to the simulated results. Nevertheless, the presented approach requires the knowledge of the exact timing of each instruction and the support of predicated instructions. Thus, a reliable result of the static analysis may only be given when the underlying processor is predictable and deterministic.

A proposal for a concrete implementation of a predictable VLIW processor is presented by Schöberl et al. in [SSP+11]. The processor is called *Patmos*, referring to the name of a Greek island. Currently, Patmos implements a dual-issue 32-bit VLIW architecture to increase the performance of single threaded execution. It is planned to add multi-chip support in the future. There are no pipeline stalls except for instructions waiting on the memory controller. All delays are visible at ISA level, which allows reliable WCET analysis, but also needs sophisticated scheduling techniques to guarantee exact and efficient execution.

The instruction set of Patmos is based on common RISC architectures like MIPS. All instructions may be predicated; the final number of different predicates is not yet fixed, but will be at least 8. There are two identical pipelines, which consist of three stages each and support operand forwarding. Both pipelines have access to the same register set. To still increase performance, Patmos has several kinds of caches, namely a method and a stack cache like JOP, a data cache and a scratchpad memory.

The original idea behind VLIW processors was to move complexity from hardware, like out-of-order execution and rearrangement, to software, meaning that a compiler is responsible for the correct scheduling of instructions. The Patmos architecture extends this approach such that the compiler is also responsible for finding out worst-case execution paths and rearranging instructions. A conventional compiler is responsible for identifying hot spots in the control flow graph of the program and optimizing them. A WCET aware compiler should be able to take annotations by the programmer in the source code into account and to find solutions for worst-case scenarios. Unfortunately, this is still an area of research and no general applicable solution has been found so far.

A first prototype of Patmos has been implemented on an FPGA. Moreover, a backend for

the LLVM compiler [LLV11] based on backends for similar architectures, has been written. A porting of the GNU binutils library for Patmos is planned. Unfortunately, the approach is still under development such that no evaluation results exist up to now. Nevertheless, due to the scientific research on VLIW architectures by Yan and Zhang, it can be expected that Patmos will offer quite efficient performance and predictable timings. In every case, time predictable VLIW processors for real-time systems are still at an early stage, but may be an interesting alternative to existing processors.

3.10 Conclusion

The current chapter enumerated some interesting approaches of the last two decades in the field of time predictable architectures. Of course, it is not possible to present all feasible solutions, but the described processors cover a wide area of possibilities. Nearly all approaches tried to provide exact execution times for each instruction such that WCET analysis of given programs becomes easier. Moreover, different techniques to increase performance and preserve predictability have been applied, ranging from pipelines shared among concurring tasks to scratchpad memories and predictable method caches. Unfortunately, it is nearly impossible to compare the presented processors, because they are based on completely different architectures and the authors used distinct benchmark suits for the evaluation of their processors. Most interestingly, most ISAs of the presented processors are based on given instruction sets and were only slightly adapted. The impact of different instruction sets on static WCET analysis is still an open field of research.

CHAPTER 4

Extensions and Modifications of an Existing Instruction Set

4.1 Problem Statement

Although nearly all processors and architectures of Chapter 3 try to provide features necessary in hard real-time systems, none of the presented solutions focus on the impacts of the implemented instruction set. The main goal is to find a useful hardware solution such that the resulting instructions are highly predictable concerning timing analysis. Nevertheless, most approaches just take the given instruction set of an existing processor without reflecting whether these instructions provide all functionality required in the context of real-time applications.[1]

The main target of this thesis is to explore the impacts of certain instructions which are added to the existing instruction set of a general purpose processor. In the subsequent chapters, the following questions are tried to be answered:

1. Do the instruction set extensions simplify the WCET analysis, e.g., by reducing branches?
2. How difficult is it to add these instructions to an existing instruction set?
   2.1. How may the additional instructions be integrated into existing opcodes?
   2.2. How may the additional instructions be implemented as hardware modules?
3. Which impacts do the instruction set extensions have on existing code generators?
4. How do the additional instructions influence the timing behavior of given algorithms?
5. Which additional instructions may be considered to be useful and which combinations of additional instructions turn out to be most effective?

[1] One exception is the PRET architecture by Edwards and Lee, which provides a *deadline* instruction to guarantee predictable timings. Confer Section 3.8 for details.

Although not all of the given questions can be answered in full detail, the presented approach tries to give a good orientation for future design considerations concerning the ISA of time predictable processors. The evaluation is based on a simulator of a SPARC V8 processor in combination with an extended code generating backend of the LLVM compiler framework. The reason for the choice of this combination is based on the following reflections:

(a) The SPARC V8 processor is a commonly used 32-bit general purpose processor, which does not provide any exceptional instructions.
(b) There exist open source implementations of the SPARC V8 architecture like the LEON2 processor. In a future evaluation, the presented instruction set extensions of this thesis may also be implemented and verified in hardware.
(c) Extending an existing code generator like GCC or LLVM allows evaluating the impact of different implementations of the same algorithm in a high-level programming language on the generated assembler code. The results may easily be compared, because no manual optimizations have been added.
(d) The LLVM compiler framework provides a code generation backend for SPARC V8 processors which might be easily extended by additional passes to produce assembler output with new instructions.
(e) Implementing an assembler and a simulator allows evaluating algorithms which would be far too complex for a manual timing and code size analysis.

The workflow of the evaluation of most presented algorithms in the current thesis was done the following way: First, a given algorithm was implemented in the C programming language. The code was translated by `llvm-gcc` into the LLVM intermediate representation (IR) language, which may be seen an equivalent to JVM assembler code. Afterwards, the LLVM compiler translated the IR code into SPARC V8 assembler code or SPARC V8 assembler including several of the presented instruction set extensions. An assembler created a binary file, which serves as input for the simulator. Several useful instruction set extensions have been combined to form a new SPARC V8 *target*. The assembler as well as the simulator have been implemented separately for each target. The assembler and simulator of each target provide special instructions to clear and print out an internal cycle counter in order to evaluate the performance of the resulting code.

The remaining part of the current thesis focuses on the following points: Section 4.2 gives an overview of the SPARC V8 architecture and assembler instructions. Section 4.3 introduces several additional instructions to the original SPARC V8 instruction set. It will be shown what the binary opcode of these instructions may look like and how they may be realized in hardware.

Chapter 5 explains how an existing compiler can be modified to support code generation for the newly introduced instructions. Section 5.1 gives an overview of the used LLVM compiler framework; the remaining sections of that chapter explain the specific implementation of the code generating passes for the additional instructions.

Chapter 6 presents a number of algorithms which are translated into assembler code using different combinations of instruction set extensions. The performance and code size of the resulting code is evaluated and the most feasible instruction set extensions are identified.

4.2 The SPARC V8 Architecture

4.2.1 General Features

The **S**calable **P**rocessor **ARC**hitecture is an ISA specification which may be implemented by any processor. The eighth version (V8) was introduced by the SPARC International cooperation in 1992; confer [SPA92]. It defines the instruction set and the corresponding opcodes and gives suggestions for a possible assembler language. Moreover, it gives implementation guidelines for subsets of the instruction set and specific hardware realizations. This might be the reason why there are several different implementations of the SPARC ISA. Nevertheless, as the SPARC V8 is no official standard, these implementations might not be fully compatible to each other.

The SPARC V8 is a 32-bit based RISC and uses a *register windowing* mechanism, meaning that each function possesses its own view of the integer register set: The first eight registers – `g0` to `g7` – are globally available and may be compared to general purpose registers of a common processor. `g0` is usually implemented to be read only and set to the constant value zero. Registers 16 to 23 or `l0` to `l7` (`l` is the letter "l") are the local registers of each function, meaning that they are only visible in the current window. Registers 8 to 15 are the *output* registers of the current window, registers 24 to 31 the *input registers*. In assembly language, they are usually notated as `o0` to `o7` and `i0` to `i7`, respectively. If parameters shall be passed to a function, the calling function only has to use its local output registers, which become the local input registers of the called function. Return values are passed to the calling function the other way round. The so-called *current window pointer* is a counter which saves the number of the currently active window. It is accessible through the processor state register and may be incremented and decremented by the `save` and `restore` instructions.

Although the described window mechanism has the big advantage that there do not exist any caller or callee saved registers, there are two major drawbacks: First, the number of real registers in hardware is quite high, namely eight global registers plus 16 times the number of register windows.[2] Secondly, in case of a large call stack, the available windows might not be enough to hold the registers of all calling functions and the current window pointer may finally point to a window which is already in use. Although this might be a rare scenario for most cases without recursions, an exception routine for both *window overflow* and *window underflow* has to be provided by the programmer. These routines usually are responsible for saving the register contents of the last window on the stack or to restore them. Of course, this exception mechanism has to be part of a profound WCET analysis for hard real-time systems.[3]

4.2.2 Instruction Set[4]

As already mentioned, SPARC V8 is a 32-bit architecture and in contrast to, e.g., the ARM instruction set, all instructions have four bytes. The available instructions may be divided in six categories: Load/store, arithmetic/logic/shift, control transfer, read/write control register, float-

[2] 8 local registers for each window and 8 input and output registers, respectively. As the input and output registers are shared among neighboring windows, only 8 registers are needed for each window.

[3] Confer [SPA92, p. 23-30].

[4] Confer [SPA92, p. 11ff. and p. 81-89].

ing point operate and coprocessor operate. A manufacturer of a SPARC processor is free to implement the floating point and coprocessor instructions, which saves hardware and development costs if they are omitted.

Nearly all instructions have two source operands and one destination operand. The second source operand may be a 13-bit signed immediate, all other operands are registers. If the instruction is a load or store operation, the source operands are used to calculate the memory address. The first source operand determines the base address, the second operand is used as offset. If `g0` is used as destination register, the result of the current operation is not saved, just the condition flags of the processor state register might change. Such, it is possible to implement compare or test instructions by using `subcc` (subtract and change condition flags) and `orcc` (logical or and change condition flags).

The SPARC V8 instruction set provides arithmetic and logical instructions of every kind, including multiplication, division and uncommon instructions like `xnor` (negated exclusive or) or `nor` (negated or). Some of them exist in two different versions; one sets the corresponding control flags of the processor state register, e.g., the overflow flag, the other leaves them untouched. Hence, it is possible to implement operations for 64-bit operands or provide compare and test operations as has previously been shown. The flags of the processor state register are used for conditional branches. As all branch and jump instructions take two cycles to complete, the instruction subsequently following either has to be independent from the branch condition or has to be a `nop` instruction. This is due to the fact that the instruction will be executed regardless whether the branch is taken or not. In case a `nop` has to be inserted, it is called a *delay* slot, because its only purpose is to delay the execution of the next instruction. The SPARC V8 instruction set also provides the possibility to conditionally execute the instruction following a branch: If the condition is satisfied, the result of the operation will be ignored. Although this feature might be very attractive to reduce average case performance, it complicates the WCET analysis and therefore should be avoided.

Apart from arithmetic-logic operations, the SPARC V8 ISA also specifies instructions designated for operating systems. These include an atomic register-memory swap operation or instructions that are only available in privileged mode. Moreover, there are several instructions to handle traps, which might be seen as alternative to exceptions. They are executed if exceptional conditions are fulfilled, like, e.g., when a load operation tries to access an unaligned address.

4.2.3 Binary Opcode Considerations

In [SPA92, p.43-47], three different binary formats are defined, which are used depending on the type of operation. They are determined by the two most significant bits.

(1) The first format is only used by the `call` instruction. The remaining 30 bits determine the 32-bit aligned destination address of the next program counter relative to the current program counter. As all instructions need 4 bytes, it is possible to address up to 4 GByte of instruction space.

(2) The second format is used by the `sethi` instruction and by all kind of branch instructions. As both of them only have two operands, there is enough space for saving 22-bit

immediate values. Consequently, branch instructions allow jumps of 8 MByte relative to the current program counter. This is far beyond the needs of most common scenarios.

(3) The third format is used by all arithmetic-logic instructions as well as by all instructions involving data memory access. All three operands need five bits to address the corresponding registers. In case the second source operand is an immediate value, a 13-bit signed integer may be used, providing a range from -4096 to $+4095$.

Figure 4.1 shows the implementation details of all three formats: Formats (2) and (3) use bits 25-29 to address the destination register or to encode the branch condition in case of control flow instructions. Bits 22-24 determine the instruction type in case of format (2), bits 19-24 in case of a format (3) instruction. As there are only 5 different instructions based on format (2), there are 3 unimplemented opcodes, which could be used by additional instructions. In format (3), bit 13 defines whether the second source operand is an immediate (bit is one) or a register (bit is zero). The address of the first source operand is saved in bits 14-18.

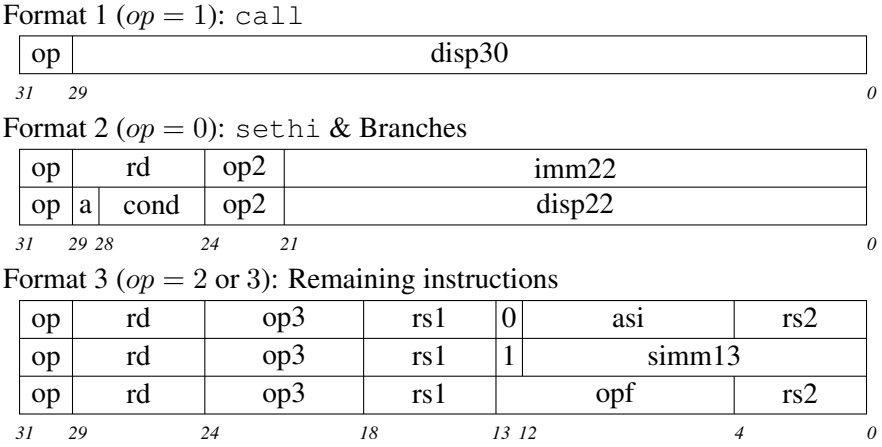

Figure 4.1: Opcode formats of SPARC V8. The figure was taken from [SPA92, p. 44]. Refer to the SPARC V8 manual for further explanations.

4.3 Instruction Set Extensions for the SPARC V8 Processor

There exist several approaches for extending instructions sets of existing processors. Some of them follow a dynamic selection strategy that decides which extensions are most effective for the given input codes. In [VATJ06], the authors use the instruction set of a PowerPC 405, which does not implement the full ISA of the PowerPC. As their target processor is implemented as an FPGA soft-core, it is relatively simple to provide additional hardware modules implementing the unavailable instructions. The selection algorithm for the instruction set extensions is based on a formula which calculates the possible speedup in case the additional operations are used. Most

of these extensions are related to floating point operations which would otherwise be emulated by software. Although their experiments were not performed in the context of real-time systems, the authors could show that it is possible to provide a complete framework which automatically selects hardware modules and instruction set extensions, to achieve better performance.

Yu and Mitra also presented an approach with instruction set extensions in [YM05], but they focus on improving WCET performance. In contrast to Veale et al., they do not use a processor implementing a subset of an ISA. They rely on soft-core processors like the Altera Nios or extensible processors like the Tensilica Xtensa, which is presented in Section 2.5. This allows them to find out the critical path within the control flow graph of a given code and to implement a critical instruction in hardware instead of software. Of course, this alternation of the resulting assembler code also influences the timing behavior and may reveal a new critical path or another worst-case scenario. Yu and Mitra used a heuristic approach for choosing which instructions are going to be implemented in hardware. The resulting problem was formulated as an integer linear program. The authors could show that the WCET performance was improved by up to 39 % in comparison to the original solution.

Beside the dynamic extension of given instruction sets, there also exist evaluations for static modifications. Two examples using the SPARC V8 architecture were realized by Tillich et al. at Technical University of Graz. In [TG06, GTS07], they tried to improve the performance of standard encrypting algorithms like AES by implementing commonly used operations as hardware instructions. The additional instructions have to be inserted manually by inline assembler. The authors showed that the execution time as well as the resulting code size of the encryption algorithms could be reduced rigorously.

One major target of the SPARC V8 instruction set extensions described in this thesis is to simplify the WCET analysis of given algorithms, e.g., by reducing branches, and to improve the temporal predictability. Nevertheless, the resulting average and worst-case performance, code size and implementation effort for hardware modules must not be neglected. The following subsections present the syntax and semantics of the additional instructions, their possible opcode encoding and suggestions for a hardware implementation.

4.3.1 Conditional Move and Conditional Select Instructions

The idea to provide a conditional move or a conditional select instruction is based on multiple reflections: First, this kind of instruction is the minimum requirement to transform existing algorithms into a single-path variant. Second, the SPARC V9 instruction set provides a conditional move instruction. As the used LLVM compiler framework also provided passes to generate assembler output for SPARC V9 targets, these instructions only had to be slightly adapted. Third, the LLVM intermediate representation (IR) language provides a conditional select (`selcc`) instruction, meaning that the translation process may be implemented easily. Moreover, many current processors support conditional moves (refer to Chapter 2), indicating that it is not excessively complicate to implement them in hardware.

Figure 4.2 shows the assembly language syntax used in the current thesis. In contrast to the SPARC V9 conditional move instruction, it is only possible to specify integer condition codes (floating point operations have not been considered at all). If the specified condition is met, the contents of the source register are copied to the destination register. Otherwise, it is left

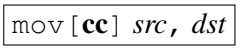

Figure 4.2: Conditional move instruction: **cc** is replaced by a condition code, *src* and *dst* represent the source and the destination register, respectively.

untouched. Such, it is possible to remove conditional branches in case of if-then-else structures as shown in Code Example 4.1. In most cases, the compiler recognizes this kind of selections automatically and uses the corresponding instruction of the LLVM IR language. Thus, the assembler code generator only needs to replace the select instruction with the newly introduced SPARC V8 instruction. A possibility to enforce the compiler to make use of conditional move or select instructions is to write the code in a SSA (single static assignment) similar manner as has been done in the comments of the SPARC assembler in Code Example 4.1.

Code Example 4.1 A simple if-then-else construction in C may be translated without using conditional branches when a `movcc` instruction is available.

```
if (a == b) {
  c = 1;
} else {
  c = 2;
}

or      %g0, 1, %i0     ! c1 = 1;
or      %g0, 2, %i3     ! c2 = 2;
subcc   %i1, %i2, %g0   ! cc = (a == b);
mov[ne] %i3, %i0        ! c = cc ? c1 : c2;
```

To implement the conditional move instructions into the existing opcode of the SPARC V8, the unimplemented opcodes of format (2) were used: As can be seen in Figure 4.1, there are three bits available for the *op2* field, resulting in eight different instructions of which five are currently in use. Thus, up to three additional instructions can be defined. For the evaluation of the benchmarks within this thesis, the opcode shown in Figure 4.3 was used for the conditional move instruction. The bits for both source operands and the integer condition codes conform to opcode format (2) and (3): The two most significant bits are set to 0 to identify a format (2) instruction. The *op2* field is set to the constant value 3 in decimal representation as this value is currently not used by any other instruction. Bits 25 to 28 are used for the integer condition codes like in conditional branch instructions. As the conditional move instruction in the presented variant is not very complex, it would also be possible to implement checking the condition codes of the floating point or of the co-processor.

The hardware implementation of the conditional move instruction is quite simple, because it only needs additional logic elements for the instruction decoding stage and a multiplexer for the selection of the corresponding register. Figure 4.4 shows a possible realization in hardware. The presented approach does not rely on a specific pipeline model, but it is assumed that the opcode has previously been decoded – indicated by the dashed line on the left hand side – and that there

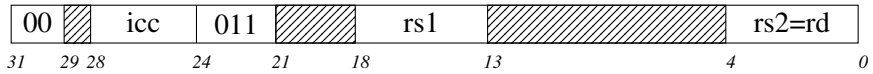

Figure 4.3: Opcode proposal for the conditional move instruction. Unused bits have been crossed out.

is some write back mechanism which saves the result in the corresponding register. The integer condition code (*icc*) of the conditional move is evaluated with respect to the currently set flags of the processor state register (*psr*). The output of this operation (*sel*) serves as select signal for the multiplexer. The result and the address of the destination register (*rd*) is forwarded to the next pipeline stage, which might be implemented by another hardware module.

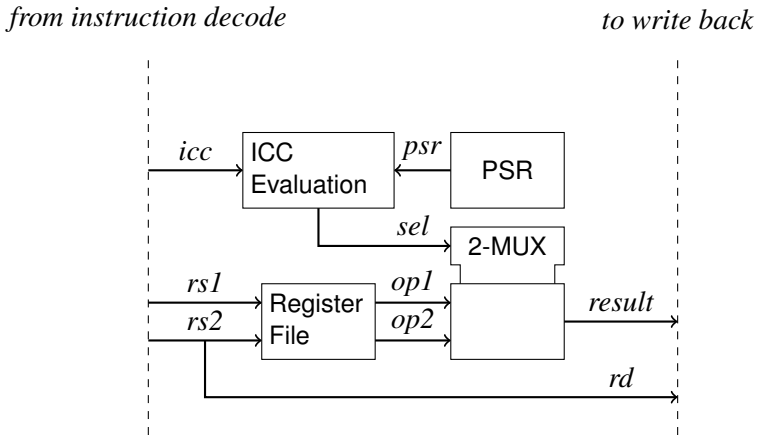

Figure 4.4: Possible block layout of the conditional move instruction.

As has already been mentioned, the LLVM IR language provides a conditional select instruction which also allows us to use signed immediate values for both source operands. Consequently, translating them to conditional moves could involve a number of additional copy instructions. The opcode of the conditional move instruction presented in the current thesis does not need all available 32 bits. These are the reasons why an alternative instruction has also been implemented: the conditional select instruction. The assembler code syntax is shown in Figure 4.5: Depending on the condition code and the set flags of the processor state register, the value of one of the two source operands is copied to the specified destination register. In case the condition is met, the first source operand will be moved, otherwise the second one. In contrast to the conditional move instruction, the second source operand does not need to be identical with the destination register. Moreover, both source operands may be signed immediate values.

Code Example 4.2 shows the resulting assembler code when using a conditional select instruction. The underlying C code is identical to the one used in Code Example 4.1. Note that the version involving conditional moves needs two additional initialization instructions to copy the

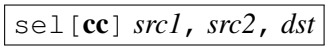

Figure 4.5: Conditional select instruction: **cc** is replaced by a condition code, *src1* and *src2* may be a register or a signed immediate value and *dst* is the register, the selected value is saved to. In the presented implementation, it is not possible to use an immediate value for *src1* and a register for *src2* at the same instruction.

immediate values into temporary registers. These steps may be omitted for conditional selects if the immediate values do not exceed the boundaries presented in the upcoming paragraphs.

Code Example 4.2 A simple if-then-else construction in C may be translated without using conditional branches when a `selcc` instruction is available. Moreover, in simple cases, there is no need to copy immediate values to temporary registers.

```
if (a == b) {
  c = 1;
} else {
  c = 2;
}

subcc   %i1, %i2, %g0   ! cc = (a == b);
sel[e]  1, 2, %i0       ! c = cc ? 1 : 2;
```

There are three different types of the conditional select instruction: The first type is used when both source operands are registers, the second type when the first source operand is a register and the second operand an immediate value and the third type when both source operands are immediate values. As can be seen in Figure 4.6, nearly all available bits could be exploited. Moreover, the boundaries for the immediate values of both source operands are given: If the first source operand is a register, the second source operand may be an immediate value from -1024 to $+1023$. If both source operands are immediate values, they can only be from -128 to $+127$. Note that there is one remaining bit combination for the type field ($ty = 3$). It could be used if the first source operand is an immediate value while the second is a register. This variant has not been implemented due to the facts that it needs additional hardware and that it does not provide additional functionality.[5]

Although the introduction of the conditional select instruction might not involve the introduction of additional complex hardware modules at the first glance, the instruction decoding is more sophisticated than for other instructions: The used bits for the first source operand register (*rs1*) differ from format (3). Moreover, the instruction introduced two new data types, namely signed immediates with 8 and 11 bits. Finally, the integer condition codes (*icc*) are saved in another bit field than it is the case for format (2) instructions. Unfortunately, there does not exist an ideal solution for this encoding problem, because other implementations would introduce

[5] If the first operand has to be an immediate value, the source operands can be switched and the selection condition has to be negated. This could easily be done by the compiler or the assembler.

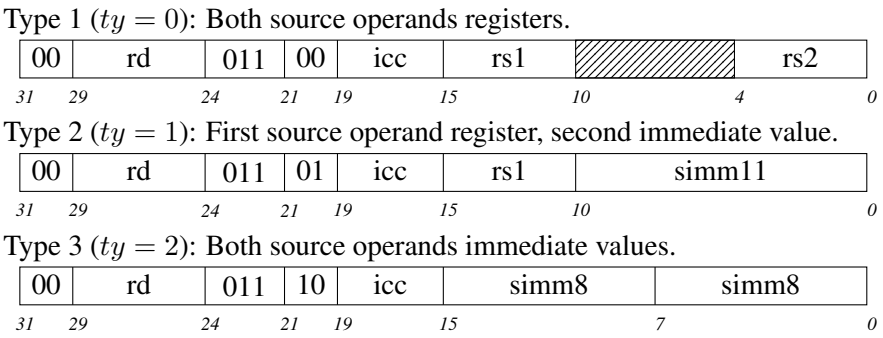

Figure 4.6: Opcode proposal for the conditional select instruction. Unused bits have been crossed out. The type field (bits 20-21) determines the type of the source operands.

complexity for other bit fields. Thus, extending an existing ISA by a conditional select instruction requires far more additional hardware than extending it by a conditional move instruction. Whether this additional hardware cost can be justified will be analyzed in Chapter 6.

4.3.2 Predicated Instructions

Predicated or conditional instructions are based on the idea that an operation is only executed if a certain condition is met. A most basic variant is available in many instruction sets: a `skipcc` instruction. It means that the subsequent instruction is only executed if the specified condition code applies. Another type of predicated instruction has just been presented: the conditional move and conditional select. ISAs which only provide a few predicated instructions are called *partially predicated instruction sets*. If every instruction is executed based on a specified condition, the ISA is called *fully predicated instruction set* as defined in [MHM+95].

Predicated instruction sets have been known since the first computers, but have not been in use for a long time. At the beginning of the 1990's, when instruction-level parallelism (ILP) was a current topic of research, predicated instructions were subject of compiler generators again: In [PS91], Joseph Park and Mike Schlansker explain how a predicated instruction set may help scheduling algorithms for parallel instruction execution. The main idea is that two disjoint paths of the control flow graph may be executed in parallel because they do not depend on each other. Unfortunately, the hardware of a processor cannot easily determine which paths are disjoint. However, if the paths may be identified by mutually exclusive conditions specified for each instruction, both paths may be easily identified. In [MHM+95], Mahlke et al. show the difference between a partially and a fully predicated instruction set and the impact on code generators, path analysis and instruction level parallelism. Nevertheless, most research in the field of predicated instruction sets has not been done in the context of WCET analysis and real-time systems.

In principle, there exist two possibilities to implement the conditions or predicates for a fully predicated instruction set: Each instruction is executed conditionally based on condition codes or based on a separate *predicate register*. Both variants exist and have been implemented; the

first one is used in the ARM instruction set, the second one is implemented by the ISA of the Intel Itanium processor. Code Example 4.3 shows the difference for a SPARC V8 instruction set extension: The first variant is based on condition codes, the second one on a predicate register. To save the current condition to a specified register, an additional `predset` instruction is necessary. Although the first variant seems to be more efficient at the first glance, the second one is more flexible when dealing with nested conditions, because they can easily be saved to a predicate register. Unfortunately, providing a fully predicated instruction set requires additional bits for each instruction, which have to be saved in the opcode. Of course, this code size overhead may be compensated by the compiler by saving branch instructions etc. Howsoever, this involves adaptations to existing compilers and may be another reason why many current ISAs do not provide a fully predicated instruction set.

Code Example 4.3 A more complex if-then-else block may be easily translated to assembler code if the instruction set supports a fully predicated instruction set based on integer condition codes (upper assembler code) or on a separate predicate register (lower assembler code). The predicate registers may be recognized by the leading *p* (e.g., `%p0`). The condition, when an instruction may be executed, is given in the square braces. For the second variant, an additional flag is needed to check whether the predication register has to be set (flag is `t`) or cleared (flag is `f`).

```
if (a == b) {
  c = a + b;
} else {
  c = a - b;
}

subcc   %i1, %i2, %g0
add[e]  %i1, %i2, %i0       ! c = (a == b) ? (a + b) : c;
sub[ne] %i1, %i2, %i0       ! c = (a != b) ? (a - b) : c;

subcc       %i1, %i2, %g0
predset[e]  %p0
add[%p0][t] %i1, %i2, %i0   ! c = (a == b) ? (a + b) : c;
sub[%p0][f] %i1, %i2, %i0   ! c = (a != b) ? (a - b) : c;
```

Although the LLVM compiler has been adapted to code generation of fully predicated instructions within the context of this thesis, the current implementation of the assembler and the simulator do not support them. This is due to the huge amount of adaptations which are necessary when the current opcode formats of the SPARC V8 should support fully predicated instructions. Nevertheless, a solution to evaluate the principle of predicated instructions has been found and is presented in the following section.

4.3.3 Predicated Blocks

A possibility to extend the existing SPARC V8 instruction set to support predicated instructions is to introduce so-called *predicated blocks*. The idea is based on the ARM thumb instruction set,

which offers an instruction predicating the following four instructions (see Assembler Code 2.5 in Section 2.2.5.). To be more flexible and provide the functionality of a fully predicated instruction set, the following instructions have been introduced: When providing predicated instructions based on integer condition codes, there is only need for a *begin* and an *end* instruction. When dealing with predicate registers, one instruction to set and one to clear them is needed additionally.

```
predbegin[cc]
predend
```

Figure 4.7: Predicated blocks based on integer condition codes: All instructions following a `predbegin` instruction will be conditionally executed based on the condition code specified in the **cc** field.

The syntax of both needed instructions for the first variant is given in Figure 4.7: A predicated block always begins with a `predbegin` instruction and may be either ended by another `predbegin` or by a `predend` instruction. There are some restrictions on the instructions which might occur within predicated blocks: An instruction setting the flags of the processor state register may cause undefined behavior. Although it seems quite reasonable that there is no real need for a compare instruction to be within a predicated block, the flags of the processor state register are also touched by an add operation of two 64-bit integers. This restriction is irrelevant for predicated blocks relying on predicate registers as will be described later. Furthermore, branch instructions of any kind should be avoided.

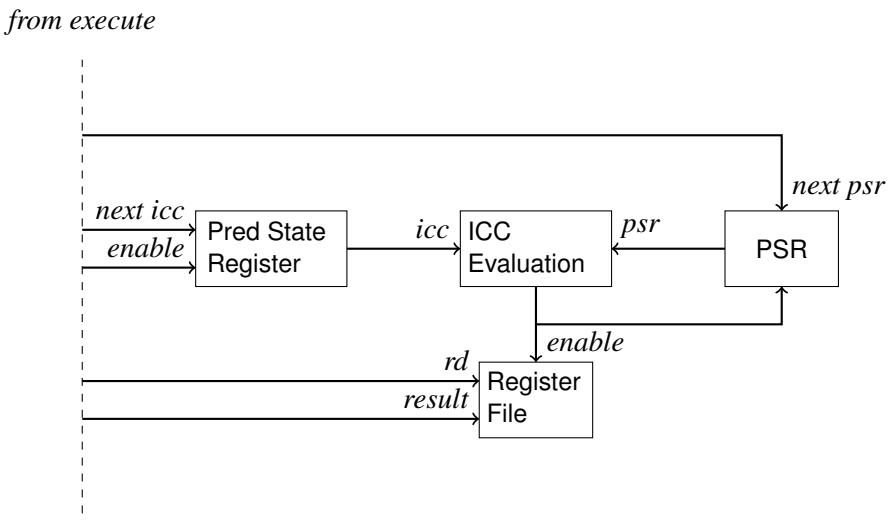

Figure 4.8: Possible block layout of predicated blocks based on integer condition codes.

The reason for this can be seen in Figure 4.8, which shows the block diagram of a possible

hardware implementation: It is assumed that the result of the current operation and the address of the destination register are available, e.g., from a preceding execute stage. They are denoted *result* and *rd* in the figure. If the current instruction is `predbegin`, the condition codes for all following instructions have to be saved. This is done by the two signals *next icc* and *enable*. A `predend` instruction may easily be implemented by setting the *next icc* signal to *always*. Based on the *icc* signal and the current processor state register (*psr*), an *enable* signal is generated which controls whether the *result* and the new value for the processor state register (*next psr*) will be written. To guarantee that all instructions are executed by default, the *icc* signal has to be set to *always* initially.

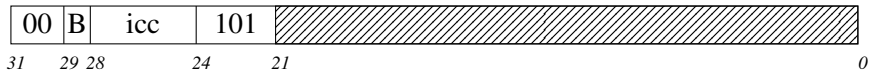

Figure 4.9: Opcode proposal for predicated blocks based on condition codes. Unused bits have been crossed out.

Figure 4.9 shows the opcode format for the predicated block instructions based on condition codes: *op2* is set to the constant decimal value 5, which is currently not in use by any SPARC V8 instruction. The integer condition code of the `predbegin` instruction is saved in bits 25 to 28, which is conform to the format of conditional branch instructions. Bit 29, labeled *B* in the figure, is used to determine whether it is a `predbegin` ($B = 1$) or a `predend` ($B = 0$) instruction. An alternative could be to use the integer condition code *always* to finish a predicated block. Thus, `predend` and `predbegin[a]` would be equivalent.

When condition registers are used instead of condition codes, additional instructions and hardware modules are necessary. A predicate register should be capable to save at least three different states: one *true* state, meaning the predicate is set, one *false* state, indicating that a condition is not fulfilled and a *clear* state which is necessary to initialize the register and to implement nested if-then-else structures (see Section 5.3.2 for details). Consequently, two bits for each predicate are needed: A conditional `predset` instruction sets the *true* bit and clears the *false* bit if the condition is satisfied. Otherwise, the *true* bit is cleared and the *false* bit is set. A `predclear` instruction clears both bits. The instructions within a predicated block are only executed if the specified true or false bit of the predicate register as defined by the `predbegin` instruction is set (refer to the first line of Figure 4.10). Consequently, if a predicate register is cleared and is not set by the conditional `predset` instruction, neither the true, nor the false predicated block will be executed.

For the simulator, a 32-bit predicate register has been used, meaning that there are 16 different predicates available. In the assembler code, they are labeled from `%p0` to `%p15`. Although only four bits are needed to address all 16 registers, the existing five bit fields for the destination and source 2 registers of format (3) have been used as can be seen in Figure 4.11. The *rdp* field represents the number of the destination predicate register and is identical to the *rdp* specified in the assembler code instructions. The predicate source register *rsp* as needed by the `predbegin` instruction is saved in bits 0 to 4. For the conditional `predset` instruction, the condition code is saved to bits 16 to 19. The **tf** flag of the `predbegin` instruction will be saved

```
predbegin[rsp][tf]
predend
predset rdp
predset[cc] rdp
predclear rdp
```

Figure 4.10: Predicated blocks based on predicate registers. *rsp* and *rdp* are predicate registers, representing the condition (*rsp*) or the destination (*rdp*) of an instruction. **tf** may be either `t`, indicating that the corresponding predicate register has to be set, or `f`, if the predicate register has to be cleared. **cc** is the condition code upon which the predicate register will be set.

in bit 5. To distinguish between the different predicated block instructions, the type field is used: for `predbegin`, *ty* is 0, for `predend`, *ty* is 1, and for `predset` and `predclear`, *ty* is 2. An unconditional `predset`[6] is indicated by the condition code *always*, a `predclear` by the condition code *never*. In all other cases, the condition code is evaluated and the corresponding bits of the predicate registers are set. Code Example 4.4 shows the practical use and the difference between both predicated block variants.

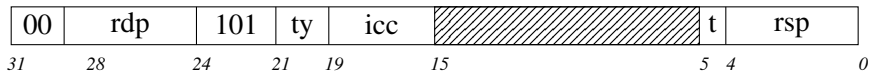

Figure 4.11: Opcode proposal for predicated blocks based on predicate registers. Unused bits have been crossed out.

Concerning a possible hardware implementation of predicated blocks with predicate registers, it has to be said that the instruction decoding is more complex than the variant only relying on integer condition codes: The type field has to be evaluated additionally and the *icc* field has another position than in any other instructions. Moreover, the involved register addresses describe another register file which also has to be accessed in an operand fetch and write back stage. Besides these concerns, the hardware implementation might be quite similar to the block layout shown in Figure 4.8. Instead of saving integer condition codes, the states of predicate registers are evaluated and the processor state register is only needed for evaluation condition codes in case of a `predset` instruction. Consequently, it is possible to have instructions touching the flags of the processor state register within predicated blocks, because the condition, when a block is executed, is saved separately.

4.3.4 Hardware Loops

Some modern processors offer special hardware loop instructions like the Blackfin and TriCore processors (see Sections 2.1.5 and 2.4.3). By such hardware loops, it is possible to have a perfect

[6] An unconditional `predset` sets both, the *true* and the *false* bit of a predicate register.

Code Example 4.4 Conversion of the same C code as used in Code Example 4.3 to predicated blocks. The first variant uses predicated blocks based on integer condition codes, the second is based on predicate registers. Note that in the second variant, the `predclear` instruction is not necessary for the presented example, but may be important for nested if-then-else structures. Although the code size is greater than that of a fully predicated instruction set, the needed modifications of hardware modules and adaptations to an existing assembler are far less complex. Beside from the code size and the resulting increased cycle count, the fully predicated instruction set provides the same features as the block variants.

```
if (a == b) {
  c = a + b;
} else {
  c = a - b;
}

subcc %i1, %i2, %g0
predbegin[e]
add   %i1, %i2, %i0       ! c = (a == b) ? (a + b) : c;
predbegin[ne]
sub   %i1, %i2, %i0       ! c = (a != b) ? (a - b) : c;
predend

predclear  %p0
subcc      %i1, %i2, %g0
predset[e] %p0             ! p0 = (a == b);
predbegin[%p0][t]
add        %i1, %i2, %i0   ! c = (p0) ? (a + b) : c;
predbegin[%p0][f]
sub        %i1, %i2, %i0   ! c = (!p0) ? (a - b) : c;
```

branch prediction of conditional jumps usually occurring at the tail of a loop. Although the ISA of the TriCore processor also allows branch instructions within a hardware loop, this solution does not seem to be adequate for WCET analysis.

In [LSMA99], Lee et al. implemented a new prediction algorithm for so-called *short backward branches* instructions. The authors analyzed many embedded programs and found out that PC-relative backwards branches within 16 instructions are one of the most common scenarios for loops. They introduced two additional instructions, one for short backward and one for short forward branches. The authors gave a proposal for an opcode implementation and showed that it is possible to implement the new instructions in hardware. Moreover, they evaluated the performance including these instructions in comparison with hardware loop instructions similar to the Infineon TriCore processor. Although their solution could improve the average performance by about 7 %, the performance when including native loop instructions was even better.

One possibility to simplify the WCET analysis of a given program is to reduce branches, which correspond to edges of the control flow graph. A hardware loop instruction which automatically decrements an index register and performs a conditional jump is an easy means to reduce the number of branches. The chosen implementation is similar to the hardware loop in-

```
hwloop init label, %loops
hwloop init label, %loope
hwloop init src, %loopb
hwloop start
```

Figure 4.12: Instruction proposal for hardware loops.

struction of the Blackfin processor: Figure 4.12 shows the assembler syntax of the implemented instructions. There are three registers which contain all necessary information of an active loop: `%loops` (loop start) saves the absolute address of the first instruction within the loop. Although this information could be automatically retrieved from the current PC value, it provides the user more flexibility and is easier to read. More efficient implementations could spare this instruction. `%loope` (loop end) saves the absolute address of the first instruction after the loop. Both are initialized with a *label*, which will be replaced with a PC-relative value by the compiler. The `%loopb` (loop bound) register saves, how many times the loop will be executed. The source operand may either be a 22 bit unsigned immediate (allowing loops to be executed up to $2^{22} - 1$ times) or an integer register containing the value. Such, it is possible to make use of greater loop bounds up to 32-bit values. Finally, there is a `hwloop start` instruction, which informs the processor that all loop related registers have been initialized correctly. Code Example 4.5 shows, how these instructions may be used to implement a simple for-loop in assembler.

Code Example 4.5 Implementation of a simple for-loop in assembler featuring native hardware loop instructions. Note that the loop body has to consist of two instructions at least. This is the reason why the `nop` instruction has been inserted after the `add` operation.

```
for (i = 0; i < 10; i++) {
  sum = sum + 1;
}
```

```
    hwloop init .loopbegin, %loops
    hwloop init .loopend, %loope
    hwloop init 10, %loopb
    hwloop start
.loopbegin:
    add    %i0, 1, %i0    ! sum = sum + 1;
    nop
.loopend:
```

Figure 4.13 shows a possible opcode implementation of the presented hardware loop instructions. Again, an unused value of *op2* of format (2) has been used. Like all branch instructions, the lower 22 bits are used to save label addresses relative to the current program counter, i.e., relative to the current instruction. Bits 25 to 29 are used to specify the loop instruction type: the first two bits may be used to identify multiple loop modules (e.g., for nested loops). They are currently not needed and set to 0 by default. The remaining three bits determine the type of the

five loop instructions: 000 indicates that the `%loops` register will be initialized, 001 indicates that the `%loope` register will be initialized and $01x$ that the `%loopb` register will be set. If the least significant bit is 0 for the latter case, the 22 bit immediate value of the opcode will be used, otherwise, the content of the register specified in bits 14 to 18 is copied. The `hwloop start` instruction is encoded 100.

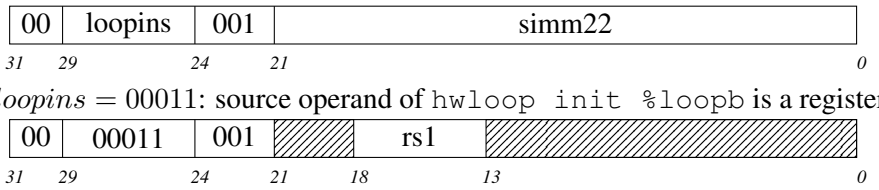

Figure 4.13: Opcode proposal for hardware loop instructions. Unused bits have been crossed out.

A possible implementation in hardware is shown in Figure 4.14: All `hwloop init` instructions save the specified values into the corresponding loop registers. They are labeled *loops* and *loope* in the Figure. The value of the loop bound is initially saved in the *loopcounter* register. The *HWLOOP State* register is set *active* by the `hwloop start` instruction. In every cycle, if the register is *active*, the value of the *loope* register is compared to the address saved in the next program counter. If they are equal, two *enable* signals are triggered: one to copy the contents of the *loops* register to the next program counter and another to decrement the loop counter. If the loop counter equals zero, the *HWLOOP State* register will be set to *idle*. As the value of the next program counter is set,[7] a hardware loop has to consist of two instructions at least. Generally said, a hardware loop instruction is just an implementation of a perfect branch prediction mechanism. Such a mechanism exists, because the loop boundaries are known at runtime and can be used to determine whether a branch will be taken or not.

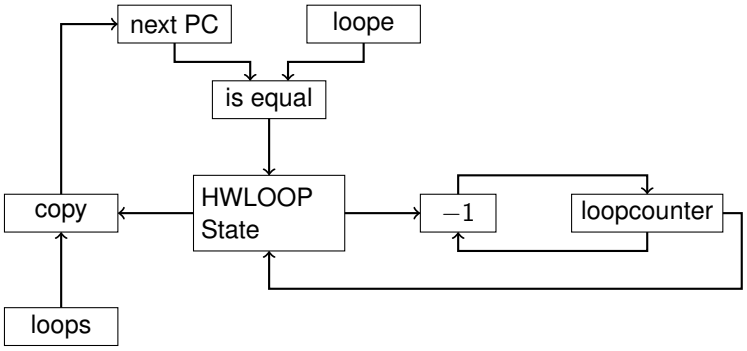

Figure 4.14: Possible block layout of a hardware loop module.

[7]The same is done by all branch instructions of the SPARC V8.

CHAPTER 5

Impacts on Code Generators

This chapter gives an overview of how the instruction set extensions presented in Section 4.3 may be added to the code generating functions of an existing compiler framework. The LLVM compiler backend has been chosen for the reasons given in Section 4.1. The structure of the chapter is the following: First, the basic principles of LLVM will be presented in Section 5.1. The subsequent sections will introduce algorithms allowing automatic code generation for the new additional instructions. These algorithms do not rely on the specifics of the LLVM compiler framework such that they may also be implemented for any other code generator. The source code of the new LLVM backend used for the benchmark evaluation of this thesis is available on GitHub (`https://github.com/cgeyer/llvm-cbg`).

5.1 The LLVM Compiler Framework

The so-called **L**ow **L**evel **V**irtual **M**achine started as a research project at the University of Illinois and was initiated by Chris Lattner and Vikram Adve. Originally, it was a tool for compiler optimization passes, but changed to a complete compiler framework over the years. In November 2003, the first stable version was released. By now (January 2012), the current stable version is 3.0, which was released in December 2011. The additional compiler passes of the current thesis were implemented on LLVM 2.9. LLVM is available under an open source license and may be freely adapted and redistributed. The main features of the first version are listed in [LA04]; all information about current versions is available online, see [LLV11].

5.1.1 Workflow for Creating Assembler Output

In [Lat11], Chris Lattner describes the basic principles of multi-phase compilers and the chosen implementation in LLVM: Usually, there is a first phase or pass which translates a high level programming language into an intermediate code representation. This representation may be subject to several optimization passes and serves as input for the backend phase. The backend phase is target dependent and creates assembler output for the specified processor. Before the

actual code is generated, there might be additional target specific optimization passes. Hence, implementing code generation for a new programming language is quite comfortable because all supported processors of the provided backend already exist as output targets. Moreover, when writing a compiler for a new ISA, only an additional backend pass has to be implemented. All supported high level programming languages may afterwards be translated into the newly added assembly language. Figure 5.1 shows the basic principle of the translation workflow when using the LLVM compiler framework.

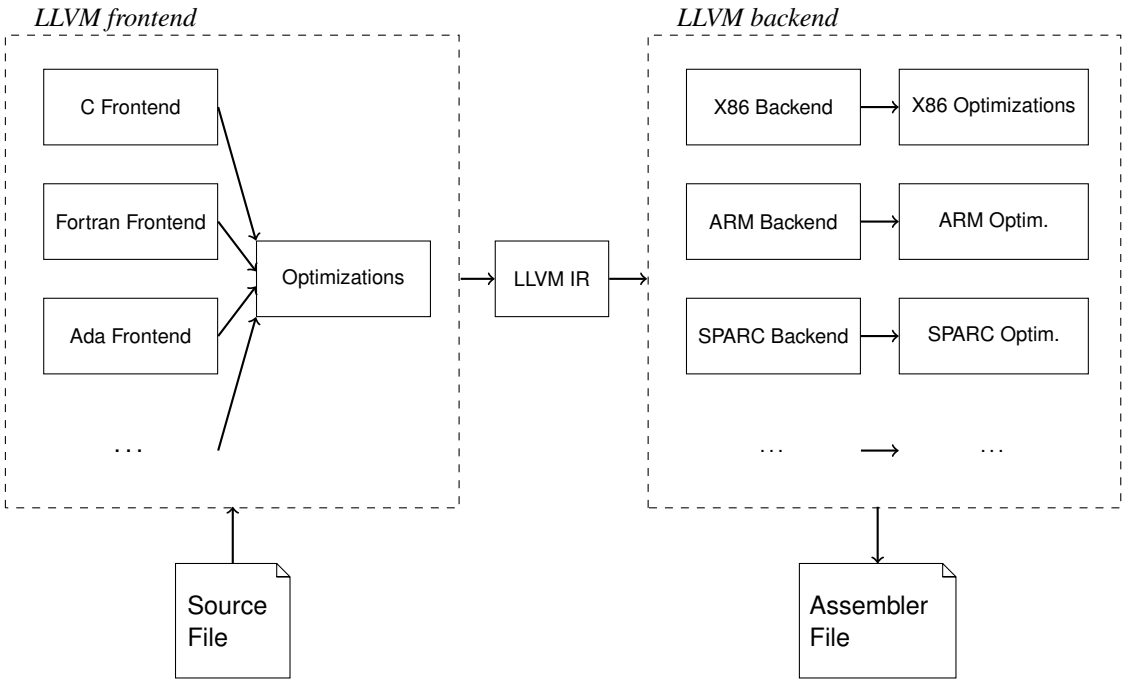

Figure 5.1: Overview of the LLVM workflow: A source file of a supported high level programming language is translated by the LLVM frontend into the LLVM intermediate representation (IR) language. This is used as input for the LLVM backend passes to create assembly or binary output for a supported target platform. The workflow sketch is based on Figures 11.3 and 11.4 in [Lat11].

One advantage of LLVM is its flexible architecture, which might be easily extended: All translation and optimization processes may be interpreted as passes which simply take a given input and might produce a new output. Dependent on the given target platform, there might be additional passes, e.g., if one implementation of a processor supports special instructions which another does not have. All these passes are summarized as *optimization passes* in the backend of Figure 5.1. Such, it is possible to provide a generic backend for one processor family, e.g., x86, and to choose the correct instructions of the corresponding target platform, e.g., an Intel Core 2 Duo processor.

The just described pass architecture of LLVM is an ideal starting point for implementing

the code generating functionality for the additional instructions presented in the current thesis: LLVM provides features to define new *subtargets* of a generic architecture such as SPARC. A subtarget only supports a subset of all available instructions. Moreover, it is possible to provide passes which will only be executed for specified subtargets. All presented code generating algorithms of the current section are based on passes which are specific for a subtarget implementing the SPARC V8 ISA with one or multiple instruction set extensions. They analyze and change the current control flow graph, insert new and delete redundant instructions. As LLVM passes are quite generic, the existing delay slot filling pass of the SPARC code generating backend is fully compatible with the newly implemented passes.[1]

5.1.2 Definitions

The current section gives an overview of the most commonly used terms when implementing LLVM passes. Although an official glossary does not exist, these definitions are based on the LLVM documentation [LLV11] and the personal experience of the author.

Frontend The LLVM frontend is responsible for translating a given high level programming language into LLVM IR language. The frontend used in the current thesis is the `llvm-gcc`, creating assembly output.

Backend The LLVM backend is responsible for translating a given LLVM IR code into assembly code of the specified target processor.

Target The Target is a concrete implementation of an LLVM backend and is usually generic. LLVM currently supports multiple targets such as x86, ARM, PowerPC and SPARC. Some are still experimental and are not officially maintained.

Subtarget An LLVM target may provide multiple features, such as floating point operations, SIMD instructions, etc. A subtarget aggregates some of these features and has to be specified for a given architecture when creating assembly language output with the LLVM backend.

Target Description File LLVM provides a generic approach to define registers, instructions and special features of a specified target. These are all saved in a target description file, which will be translated into C++ classes. Although this mechanism is quite powerful, some features cannot be described and have to be implemented natively in C++.

Basic Block A basic block in general meanings is a vertex of the control flow graph and has one defined entry and one defined exit point. In high level programming languages, basic blocks are usually delimited by special instructions like `end` or symbols like curly braces. A *machine basic block* (MBB) is a basic block in a target specific assembly language. The entry point may be either a label as defined by branch instructions or a so-called *fall-through*. This is the case when the preceding machine basic block ends with a conditional

[1] As described in Section 4.2.2, it is necessary to insert delay instructions after all kind of branches. This is done in the delay slot filling pass of the LLVM SPARC V8 backend.

jump. In case the branch is not taken, the succeeding machine basic block is entered. The exit points of machine basic blocks are all instructions which may be seen as terminators: These include all kind of branches and return operations.

Pass As has already been mentioned, the whole translating process in LLVM is implemented by several passes. There are various kinds of passes which define when they might be executed and what they are allowed to change. The most important pass for late optimizations is a *function pass*, which operates on machine functions. Machine functions consist of multiple machine basic blocks and cover all instructions of the corresponding function of the high level programming language. Although it is possible to change nearly everything within the given function, it is not possible to get access to other machine functions or to modify global variables etc.

5.2 Implementing Code Generation for Conditional Move and Conditional Select

Extending the code generator of the existing SPARC V8 to support conditional moves and conditional selects as presented in Section 4.3.1 was quite simple for the following reasons:

(1) The SPARC backend supports code generation for the SPARC V9 ISA, which provides a conditional move instruction. This conditional move has nearly the same assembler syntax as the one used in the current thesis.
(2) The LLVM IR language has a conditional select instruction, which can be easily mapped to a conditional move or conditional select instruction.

Code Example 5.1 LLVM table description definition of the conditional move instruction.

```
1  // conditional move instructions
2  let Predicates = [HasMovCC], Uses = [ICC],
3      Constraints = "$src2 = $dst" in {
4    def MOVcc_rr : Pseudo<
5      (outs IntRegs:$dst),
6      (ins IntRegs:$src1, IntRegs:$src2, CCOp:$cc),
7      "mov[$cc] $src1, $src2",
8      [(set IntRegs:$dst,
9         (
10         SPselecticc IntRegs:$src1, IntRegs:$src2,
11         imm:$cc
12        )
13     )]
14   >;
15 }
```

Hence, only small adaptions had to be implemented: The second source operand of conditional moves is also the destination operand, which may be defined by a constraint in the LLVM table description definition as can be seen in line 3 of Code Example 5.1. The conditional move

has one output register `$dst`, two input registers `$src1` and `$src2` and an immediate value representing the condition code `$cc`. The conditional move is a `Pseudo` instruction, meaning that there does currently not exist an opcode definition and that there are no special constraints, e.g., concerning the supported number of operands. Line 7 shows the syntax definition for the assembly language. All registers and immediate values will be replaced by the actual used values (e.g., `$src2` by `%i0`). The subsequent part represents the syntax tree which will be substituted by the current instruction if the specified pattern matches.

The definition for conditional *selects* is quite similar, but there exist several variants: One with two registers as source operands, two with one register and one immediate value as source operands and one with two immediate values as source operands. Moreover, the bit width of the immediate values has to be specified. The variant in which the first source operand is an immediate value and the second one is a register is not supported by the current opcode representation. Nevertheless, this operation may be easily emulated by switching both source operands and negating the condition code of the conditional select instruction. Algorithm 5.1 shows a possible implementation of this swapping function.

Algorithm 5.1 Swapping algorithm for conditional select instructions.

```
1 function swap(instr selcc) {
2   if (selcc.src1.type = immediate) {
3     change(selcc.src1, selcc.src2);
4     selcc.conditionCode := ¬selcc.conditionCode;
5   }
6 }
```

5.3 Implementing Code Generation for Predicated Instructions and Predicated Blocks

Implementing code generation for predicated instructions is quite similar to code generation for predicated blocks: The basic idea is to find machine basic blocks which will only be executed if a certain condition is met. In case of a fully predicated instruction set, that condition is added to every instruction. For predicated blocks, it is only necessary to insert a conditional `predbegin` instruction before the first predicated instruction and a `predend` instruction after the last one. Thus, identifying basic blocks which might be predicated involves the same procedure for both variants.

5.3.1 If-then-else Elimination

In [PS91], Park and Schlansker presented a branch-elimination algorithm based on predicated execution. It is based on the idea of program dependency graphs presented in [FOW87]. Their main goal was to identify and merge branches with the same predicate such that branches depending on different conditionals can be executed in parallel. This algorithm is known RK algorithm and is the basis of many other branch eliminating algorithms. Within this thesis, we will

restrict branch elimination only to some basic concepts in order to keep the additional compiler passes simple.

The basic idea of introducing conditionally executed machine basic blocks is to eliminate if-then-else branches and if-branches. Figure 5.2 shows the usual transformation of the original C code (left part) to assembler code (middle part). The condition is checked by the first machine basic block (`subcc`) which is terminated by a conditional branch (`b[cc]`). If the condition is not satisfied, the conditional jump will not be taken and the *else* block (`B`) is executed. It is terminated by an unconditional branch instruction to the first basic block after the if-then-else structure (`ba .end`). In case the condition is true, the *then* block (`A`) is executed, which is located between block `B` and the *end* block. The resulting control flow graph is shown in the right part of the Figure. The presented solution is the implemented transformation algorithm of the LLVM code generator for SPARC targets. It is the basis to reliably identify if-then-else structures in the CFG of a given program.

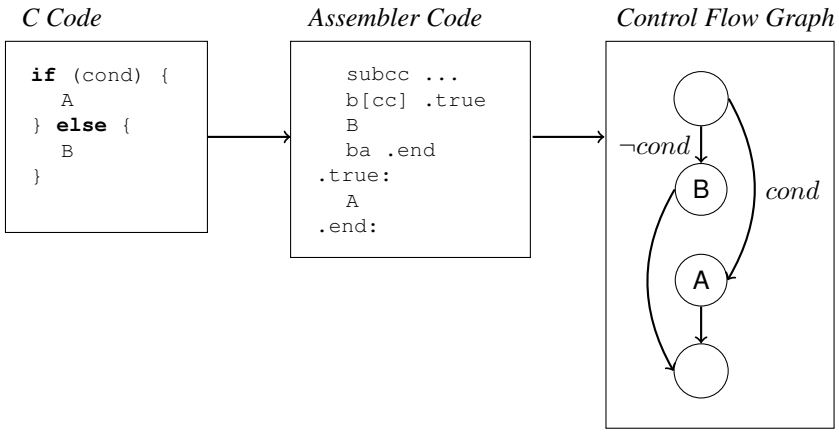

Figure 5.2: Translation process of an if-then-else structure to assembler code and the resulting control flow graph. Each vertex corresponds to a machine basic block in the assembler code.

When referring to Figure 5.2 again, an if-then-else-structure within a given control flow graph may be identified when two vertices (a *then* block `A` and an *else* block `B`) are given and the following conditions are satisfied:

(1) `A` and `B` only have one predecessor
(2) `A` and `B` have the same predecessor
(3) `A` and `B` only have one successor
(4) `A` and `B` have the same successor
(5) `A` and `B` are not identical
(6) `B` is the layout successor[2] of the predecessor of `A` and `B`

[2]A layout successor denotes the machine basic block which will be printed straight after the current machine basic block. Although this condition is not needed to guarantee correctness, it avoids that multiple combinations of the same basic blocks are identified as an if-then-else structure.

(7) `A` is the layout successor of `B`

Algorithm 5.2 shows the merging function, which is called if `A` and `B` meet the stated conditions. The `predicateBlock` function in lines 4 and 5 may either insert `predbegin` and `predend` instructions or add the corresponding condition to all instructions within the given basic blocks. The whole transformation process is illustrated in Figure 5.3. Although there are some restrictions on the used instructions within predicated blocks (no branches etc.), there is no much additional checking necessary: Branches cannot be within the machine basic blocks by definition (see Section 5.1.2), such that the code generator only has to check for `call` instructions and operations changing the processor state register.

Algorithm 5.2 Algorithm for if-then-else elimination. *MBB* is the abbreviation for machine basic block. `A` is the *then* block, `B` the *else* block. The function returns a newly created machine basic block, which contains at least the *if*, *then* and *else* blocks.

```
1  function predicate(MBB A, MBB B) {
2    condition trueCond := getCondition(A.predecessor);
3    MBB newMBB;
4    predicateBlock(A, trueCond);
5    predicateBlock(B, ¬trueCond);
6    if (|A.successor.predecessors| = 2) {
7      newMBB := A.predecessor ∪ A ∪ B ∪ B.successor;
8    } else {
9      newMBB := A.predecessor ∪ A ∪ B;
10   }
11   return newMBB;
12 }
```

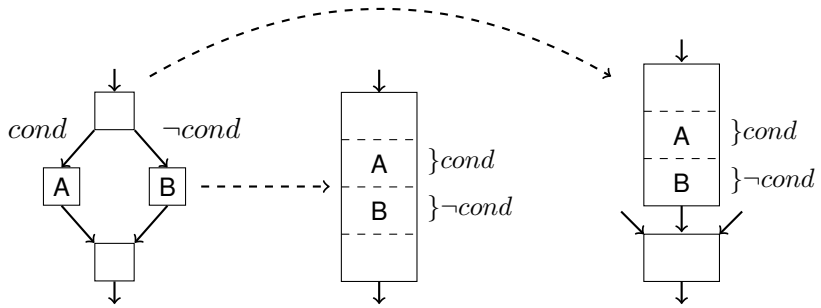

Figure 5.3: Branch elimination of an if-then-else control flow graph: The first three blocks may be merged; block `A` gets predicated with *cond* and block `B` with the opposite condition. If `A` and `B` are the only predecessors of the fourth block in the CFG, it may also be added, such that there is only one remaining block (middle part). If the fourth is the branch target of other basic blocks, it cannot be merged (right part).

Identifying a simple if-then-structure, i.e., an if-then-else-structure without an *else* branch, requires some adaptations: Let `A` be the *then* block and `B` the *end* block, i.e., the block which will be executed after `A` in any case. In this case, the following conditions have to be satisfied:

(1) `A` only has one predecessor
(2) `A` is the layout successor of its predecessor
(3) `B` is the only successor of `A`
(4) `A` and `B` are not identical

The merging algorithm for if-then-structures is quite similar to the merging of if-then-else-structures, but only block `A` will be predicated. Like before, the end block (now `B`) may only be merged if `A` is its only predecessor. The implemented LLVM pass iterates over all combinations of machine basic blocks of a given function and checks whether the conditions for if-then-else or if-then structures are satisfied. If they are, both basic blocks are passed to the corresponding predicating function. Although there might be more sophisticated variants for identifying and eliminating if-then-else structures, the presented algorithm is a correct and reliable solution and works fine in practice.

5.3.2 Nested If-then-else Elimination

Although it is possible to transform nested if-then-else-blocks to single-path variants by only using predicates based on condition codes, it is far easier to implement if predicate registers are available. They have been presented in Section 4.3.2 and are able to save the result of a compare operation. As it is possible to use a conditional `predset` instruction within a predicated block, nested if-then-else structures may be implemented easily: Code Example 5.2 shows blocks `A` and `B`, which will only be executed if `cond_1` is satisfied. Moreover, their execution also depends on `cond_2`. If the algorithm presented in the last section tries to transform the corresponding CFG, only `A` and `B` will be predicated, block `C` will still be a branch target. Nevertheless, when analyzing the resulting new control flow graph, where the if-statement, block `A` and block `B` have been merged, the algorithm can identify another if-then-else structure which may be predicated. This is only possible if the inner condition `cond_2` has been saved to a predicate register. The resulting assembler code is shown in the lower part of Code Example 5.2.

Algorithm 5.3 shows the basic principle, how nested blocks may be eliminated: If any if-then-else structure can be removed, the resulting new CFG will be analyzed again. If no branches could be removed, the iteration stops. Figure 5.4 illustrates block merging of nested if-then-else structures: In the first step, two such structures could be identified and get merged. The remaining nodes of the resulting CFG may be merged again, such that there is only one machine basic block left as shown in the right part.

Although the presented solution does not seem to be very complex, there is still one open question when dealing with nested if-then-else structures: How does register allocation work for predicate registers and when do they get cleared? Assuming, there is an iterative process for nested if-then-else elimination as shown in Figure 5.4, the first predicate register `%p0` is chosen in the first step. The register does not need to be cleared, because the corresponding true and false bits will be set in any case. In subsequent elimination iterations, the merged basic blocks

Code Example 5.2 Transformation of nested if-then-else structures to predicated blocks.

```
if (cond_1) {
  if (cond_2) {
    A;
  } else {
    B;
  }
} else {
  C;
}

subcc ...          ! Test for cond_1
predclear %p0
predset[eq] %p1
predbegin[%p1][t]
subcc ...          ! Test for cond_2
predset[eq] %p0
predbegin[%p0][t] ! if (cond_1 && cond_2)
A: ...
predbegin[%p0][f] ! if (cond_1 && !cond_2)
B: ...
predbegin[%p1][f] ! if (!cond_1)
C: ...
predend
```

Algorithm 5.3 Algorithm for nested if-then-else elimination. Line 7 makes use of an adapted version of Algorithm 5.2 to eliminate branches. As long as if-then-else structures can be removed, the resulting CFG is analyzed again.

```
1  function eliminateNestedBlocks(MachineFunction F) {
2    bool Change := ⊤;
3    while (Change) {
4      Change := ⊥;
5      for (MBB A, B ∈ F) {
6        if (isIfThenElse(A,B) ∨ isIfThen(A,B)) {
7          predicate(A,B);
8          Change := ⊤;
9        }
10     }
11   }
12 }
```

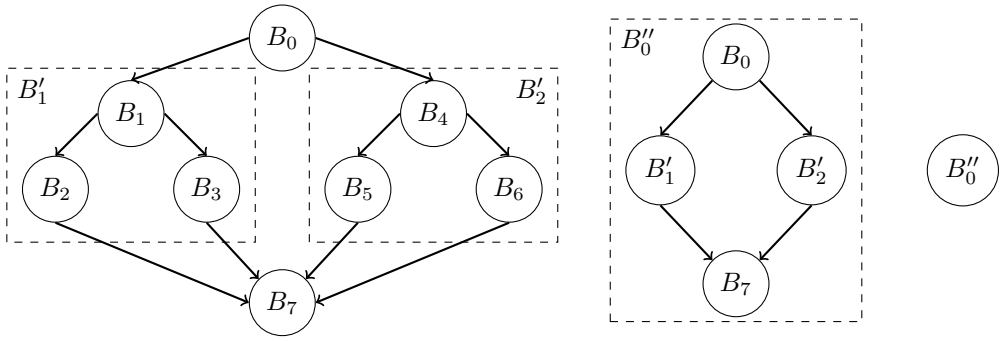

Figure 5.4: Nested if-then-else elimination: The left part shows the original control flow graph. Vertices B_1, B_2 and B_3 are merged to a predicated block B'_1 as indicated by the dashed rectangle. The transformed control flow graph is shown in the center. It may be transformed again, such that finally only one machine basic block B''_0 remains.

have to be analyzed carefully: based on the used predicate registers of the inner blocks, the next free register has to be chosen. Now, another problem arises: Assuming, there are two predicated blocks depending on the same predicate register, e.g., B'_1 and B'_2 of Figure 5.4. The original inner nodes B_1 and B_5 are both only executed if the true bit of `%p0` is set. Consequently, if B'_1 is executed and the true bit is set, the original B_5 will also be executed although the outer condition is not satisfied. One solution is to get all used predicate registers of the inner blocks and to clear them globally, i.e., in a non-predicated block before it is firstly needed.

When using predicated blocks, the most simple solution for predicating them is to insert a corresponding `predbegin` before the first and a `predend` after the last instruction. When dealing with nested if-then-else structures, it might be that multiple `predbegin` or `predend` instructions are inserted subsequently. Although this is semantically irrelevant, the runtime may be dramatically increased by instructions which are completely redundant. Hence, an additional optimization algorithm has been implemented. This algorithm is called after each if-then-else elimination iteration. It is responsible for finding multiple occurrences of `predbegin` and `predend` and simply removes all but the last instruction. The corresponding pseudo code is given in Algorithm 5.4.

Algorithm 5.4 Optimization algorithm for predicated blocks by eliminating multiple occurrences of `predend` and `predbegin` instructions within the given machine basic block.

```
1  function removePredEnds(MBB block) {
2    for (Instruction i ∈ block) {
3      if ((i=predbegin ∨ i=predend) ∧
4          (i.next=predbegin ∨ i.next=predend)) {
5        remove(i);
6      }
7    }
8  }
```

5.4 Implementing Code Generation for Hardware Loops

The basic idea of code generation for hardware loops is quite simple: Every bounded loop which is not exited early (e.g., by a `break` or `goto` instruction) shall be transformed into a hardware loop. Although LLVM has so-called *loop passes*, which automatically iterate over all basic blocks of a loop, it only allows adding and removing LLVM IR code. Thus, the code generation has been implemented as *machine function pass* nearly at the end of the compiling process. Of course, it is a simple task to find loops within the CFG of a given program, but determining the entry and exit points of the loop is far more complex. Moreover, identifying loop bounds and index variables is more difficult for multiple basic blocks or even nested loops. Consequently, the current solution only supports loops covering a single machine basic block. If the code generating pass for hardware loops is the only one active, the algorithm would be quite restricted, because conditional branches within a loop cannot be recognized. Hence, it is recommended to execute the branch elimination passes before the identification process of hardware loops. The current implementation only uses low-level information based on target dependent assembler code to identify loops, loop bounds and index variables. A better solution could include more information from the high level programming language, such that the identification process of nested loops may be easily supported.

The transformation of a given assembler code to hardware supported loops is done in multiple steps: First, all machine basic blocks which are successors of themselves are determined. Afterwards, it is tried to find a loop index candidate. This is the register which is compared with another value or register before the conditional branch at the end of the current machine basic block. The index variable has to satisfy the following conditions within the current basic block:

(1) The compare instruction including the index register must be the last operation changing the condition flags of the processor state register.
(2) The index register must only be incremented or decremented once.
(3) The index register must not be the destination register of any other instruction.

If all three conditions are met by the index register candidate, the loop bound may be found out by taking the second source operand of the compare instruction. The loop bound may be either

(1) a constant value or
(2) a register value if it is not used as destination of any operation.

It is assumed that the index variable is either incremented within $[0, loopbound)$ or decremented within $[loopbound, 0)$. Consequently, the value of the loop bound candidate register may just be moved to the corresponding hardware loop bound register. In case the conditional branch is *less than or equal*, the loop bound register needs to be incremented before it may be saved. Algorithm 5.5 shows the pseudo code implementation of the complete code generating process. The `insertHWLoop` function is responsible for creating the loop header, initializing all hardware registers as presented in Section 4.3.4. Moreover, if the current basic block is a branch target of other basic blocks, all destination labels have to be set to the newly created machine basic block. Line 6 is responsible to remove the conditional branch and compare instructions at the end of the given machine basic block. If the index register is only used to determine the loop

exit condition and not used as source operand of any operation, it can safely be removed. This is done in line 7.

Algorithm 5.5 Generating assembler code for hardware loops.

```
1  function HWLoopPass(MachineFunction F) {
2    for (MBB block ∈ F) {
3      if (block.next = block)
4        (indexReg, loopVal) := getLoopBounds(MBB);
5        insertHWLoop(MBB, loopVal);
6        removeConditionalBranch(MBB);
7        removeIndexVar(MBB, indexReg);
8      }
9    }
10 }
```

CHAPTER 6

Evaluation of Instruction Set Extensions

The last chapters presented extensions of the SPARC V8 ISA and how they may be implemented in hardware. Moreover, a prototype implementation of code generating passes for the new instructions has been described. Until now, nothing has been said about the actual performance of these extensions, i.e., whether they are useful at all and if there are some combinations of them which are more effective than other ones. Consequently, the current chapter tries to answer the following questions:

(1) Will an additional instruction be used at all by a given benchmark? If yes, how often does it occur in the resulting assembler code?
(2) How many branches will be replaced by corresponding instruction set extensions, e.g., conditional move instructions or hardware loops?
(3) How does the programming style in a high-level programming language influence the code generating process?
(4) Is the resulting code size less or greater than the original solution?
(5) Is the worst-case performance, i.e., the worst-case cycle count better or worse than the original solution?

The source code of the SPARC V8 ISA simulator including the instruction set extensions is available on GitHub (`https://github.com/cgeyer/Sparc-V8-IS-extension-simulator`).

Section 6.1 gives some basic examples how the newly introduced instructions change the CFG of a given algorithm and their influence on code size and performance. It will be analyzed what kind of high-level C-Code structures cause the LLVM compiler to introduce the new instructions and how the resulting assembler code looks like. To evaluate the best- and worst-case cycle count of the presented code examples, they will be calculated manually. Finally, it will be tried to identify useful combinations of instruction set extensions such that the resulting assembler code shows fewer branches, less code size and a better worst-case cycle count.

Section 6.2 focuses on the impact of the coding style in a high-level programming language on the resulting assembler code if various combinations of instruction set extensions are enabled: Four standard algorithms will be implemented in several variants to test whether the resulting assembler code is a single-path solution and thus shows constant execution time for all possible input data. Moreover, the code size and worst-case cycle count for multiple combinations of the instruction set extensions will be evaluated. It will be shown that there is a single-path implementation for nearly all analyzed algorithms which has a smaller worst-case cycle count in comparison with a *traditional* implementation of the algorithm in case several instruction set extensions have been enabled.

Section 6.3 assesses the general performance of the newly introduced instructions: More than 20 algorithms have been compiled and simulated on more than 10 different versions of SPARC V8 processors with several combinations of instruction set extensions. The main goal of this evaluation is to set up guidelines for future ISAs of time predictable processors by identifying combinations of instructions which show fewer branches, smaller code size and a better worst-case cycle count than the original SPARC V8 instruction set for most of the evaluated algorithms. Of course, there is no target which fits all needs perfectly. Nevertheless, most of the presented solutions show much better worst-case performance for single-path algorithms while having fewer branches than the assembler code of the original SPARC V8 target.

6.1 Manual Evaluation of Small Examples

6.1.1 Branch Elimination of Simple If-then-else Structures

One of the most common structures of usual programs are conditionally executed blocks. In a high level programming language such as C, these blocks are often implemented as if-then-else, or only if-then structures. Usually, these blocks will be translated into three different machine basic blocks on assembler level, as can be seen in Code Example 6.1: The assembler code right to the original C algorithm consists of a block checking the condition `input > 5` which is done in line 3. The original condition gets negated such that the subsequent conditional branch to label `.LBB0_2` (i.e., basic block 2) will only be executed if `input` is less than or equal to (`le`) 5. The *then* block is covered by lines 7 to 8. All in all, the solution for the original SPARC V8 instruction set generated by LLVM consists of four basic blocks, one conditional and one unconditional jump. Calculating the worst-case path reveals that block 1 consists of two instructions and consequently takes more cycles to execute than block 2.

Although this is just a trivial code example, which is not quite realistic, it shows how much influence modifications of an ISA may have on the resulting assembler code: The first improvement concerning path analysis is to introduce predicated blocks. For simplicity, we currently restrict them to integer condition codes. The assembler code shown in the left part of Code Example 6.1 only consists of one basic block and does not have any branches. Analyzing the code size shows that it has exactly the same size as the original solution, namely 40 bytes (each instruction needs 32 bits). The last presented solution introduces conditional selects, which may also take two (small) integers as source operands. Thus, the resulting code size is smaller than for the other solutions, namely 24 bytes.

Code Example 6.1 Code generation for a simple branch test. The resulting assembler code is shown for different targets: The upper right solution implements the original SPARC V8 instruction set, the lower left introduces predicated blocks and the lower right one makes use of conditional selects. The assembler code has been taken from the LLVM compiler output and has only been slightly adapted for better reading.

C-Code:

```
int branch_test(int input) {
  int tmp = 0;
  if (input > 5) {
    tmp = 3;
  } else {
    tmp = -7;
  }
  return tmp*input;
}
```

SPARC V8:

```
 1  branch_test:
 2    save %sp, -96, %sp
 3    subcc %i0, 5, %l0
 4    ble .LBB0_2
 5    nop
 6  ! BB#1:
 7    ba .LBB0_3
 8    or %g0, 3, %l0
 9  .LBB0_2:
10      or %g0, -7, %l0
11  .LBB0_3:
12    smul %l0, %i0, %i0
13    jmp %i7+8
14    restore %g0, %g0, %g0
```

SPARC V8 + `predblock`:

```
 1  branch_test:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    subcc %i0, 5, %l0
 5    predbegin[le]
 6    or %g0, -7, %l0
 7    predbegin[g]
 8    or %g0, 3, %l0
 9    predend
10    smul %l0, %i0, %i0
11    jmp %i7+8
12    restore %g0, %g0, %g0
```

SPARC V8 + `selcc`:

```
 1  branch_test:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    subcc %i0, 5, %l0
 5    sel[g] 3, -7, %l0
 6    smul %l0, %i0, %i0
 7    jmp %i7+8
 8    restore %g0, %g0, %g0
```

What about the timing behavior of the three presented solutions? All instructions of the presented assembler code take exactly one cycle, except for the multiplication, which takes 5 cycles. Obviously, the solution including predicated blocks will be the slowest, because all instructions will be executed in any case. Thus, the resulting cycle count is $9 \cdot 1 + 1 \cdot 5 = 14$. Calculating the performance of the solution with conditional selects results in $5 \cdot 1 + 1 \cdot 5 = 10$ cycles. As both solutions do not include any conditional branches, the resulting cycles will always be the same, regardless of the input value.

When calculating the worst-case cycle count of the assembler code without any additional instructions, we have to identify the worst-case path: Basic block 0 consists of four instructions, thus taking 4 cycles. Basic block 1 takes 2 cycles, whereas basic block 3, consisting of one multiplication and two simple instructions, takes 7 cycles. Thus, the resulting worst-case perfor-

mance is $4 + 2 + 7 = 13$ cycles, which is only slightly better than the solution with predicated instructions. Nevertheless, it is no single-path solution, such that the best case needs one cycle less and the path analysis is more complex. Although the presented example is somehow theoretical, it shows the basic problems of path analysis and why it is sometimes more preferable to have a less efficient solution in terms of performance. Nevertheless, the solution with the smallest code size is also easy to analyze and has the best performance. Thus, conditional move and select instructions might be very desirable instruction set extensions.

6.1.2 Branch Elimination of Nested If-then-else Structures

How does adding an additional if-then-else structure influence the code generating process? As can be seen in Code Example 6.2, the resulting assembler code for the SPARC V8 ISA now consists of 6 basic blocks and has two conditional and two unconditional branches. Identifying the worst-case path is rather difficult, but still possible: Basic block 3 may be reached by a fall through from basic block 2, which results in 5 cycles, all other combinations consume fewer cycles. Thus, if the input variable has a value from 6 to 9, the worst-case path will be taken, resulting in an overall execution time of 16 cycles. Most interestingly, the generated assembler output is identical for the conditional move and select instruction set extensions, meaning that the compiler is not able to optimize the given CFG.

However, the output for predicated blocks reveals that the implemented branch elimination algorithm presented in Sections 5.3.1 and 5.3.2 works in practice: The version using predicated blocks based on integer condition codes reduces the number of basic blocks to four. It only has one conditional and one unconditional branch. The code size is the same as for the SPARC V8, namely 60 bytes. The resulting worst-case scenario takes 17 cycles which is one more than the original code. For the variant implementing predicated blocks based on predicate registers it is possible to replace all branches such that the solution only consists of one basic block. Nevertheless, the code size is larger (72 bytes) and the worst-case performance is worse (19 cycles). Thus, reducing branches usually provokes an increase of code size and a performance degradation.

Code Example 6.3 shall demonstrate how the *coding style* may influence the code generating process: The C code is semantically equivalent to Code Example 6.2, but does not make use of if-then-else blocks. The ternary `?`-operator can be translated by the LLVM compiler, such that conditional move and select instructions are introduced. However, the generated assembler code for the original SPARC V8 ISA is very similar to the second version: It consists of six basic blocks and has two conditional branches, but only one unconditional jump. The resulting code size is 60 bytes. Finding the worst-case path is quite a challenging task because we do not only have to analyze the given CFG, but also take into account all possible values for the `input` variable. In principle, there are three different cases we have to consider:

(1) $\texttt{input} \leq 5$
(2) $\texttt{input} > 5 \wedge \texttt{input} < 10$
(3) $\texttt{input} \geq 10$.

Although the given CFG would allow the execution path including basic blocks $0, 1, 3, 4, 5$, this path is infeasible: For case (1) of all values of `input`, variable `tmp` will first be set to 4

Code Example 6.2 Slightly adapted branch test from Code Example 6.1. The nested if-then-else structure cannot be converted by LLVM, such that conditional move or select instructions are used; the resulting assembler output is equal to the output for the original SPARC V8 ISA. However, the inner branch may be eliminated when predicated blocks based on integer condition codes are used; see lower left part. When predicate registers are used instead, the resulting code is a single-path implementation as can be seen in the lower right part.

C-Code:

```
int branch_test2(int input) {
  int tmp = 0;
  if (input > 5) {
    if (input < 10) {
      tmp = 3;
    } else {
      tmp = 4;
    }
  } else {
    tmp = -7;
  }
  return tmp*input;
}
```

SPARC V8:

```
 1 branch_test2:
 2 ! BB#0:
 3   save %sp, -96, %sp
 4   subcc %i0, 6, %l0
 5   bge .LBB0_2
 6   nop
 7 ! BB#1:
 8   ba .LBB0_5
 9   or %g0, -7, %l0
10 .LBB0_2:
11   subcc %i0, 10, %l0
12   bge .LBB0_4
13   nop
14 ! BB#3:
15   ba .LBB0_5
16   or %g0, 3, %l0
17 .LBB0_4:
18   or %g0, 4, %l0
19 .LBB0_5:
20   smul %l0, %i0, %i0
21   jmp %i7+8
22   restore %g0, %g0, %g0
```

SPARC V8 + `predblock` icc:

```
 1 branch_test2:
 2 ! BB#0:
 3   save %sp, -96, %sp
 4   subcc %i0, 6, %l0
 5   bge .LBB0_2
 6   nop
 7 ! BB#1:
 8   ba .LBB0_3
 9   or %g0, -7, %l0
10 .LBB0_2:
11   subcc %i0, 10, %l0
12   predbegin[ge]
13   or %g0, 4, %l0
14   predbegin[l]
15   or %g0, 3, %l0
16   predend
17 .LBB0_3:
18   smul %l0, %i0, %i0
19   jmp %i7+8
20   restore %g0, %g0, %g0
```

SPARC V8 + `predblock` pred. reg.:

```
 1 branch_test2:
 2 ! BB#0:
 3   save %sp, -96, %sp
 4   subcc %i0, 6, %l0
 5   predclear %p1
 6   predclear %p0
 7   predset[ge] %p1
 8   predbegin[%p1][t]
 9   subcc %i0, 10, %l0
10   predset[ge] %p0
11   predbegin[%p0][t]
12   or %g0, 4, %l0
13   predbegin[%p0][f]
14   or %g0, 3, %l0
15   predbegin[%p1][f]
16   or %g0, -7, %l0
17   predend
18   smul %l0, %i0, %i0
19   jmp %i7+8
20   restore %g0, %g0, %g0
```

Code Example 6.3 Semantically equivalent implementation of Code Example 6.2, using the ternary `?`-operator instead of if-then-else structures. The code generator is now able to introduce conditional move and select instructions, resulting in fewer branches, smaller code size and better performance. The assembler output for the SPARC V8 instruction set is very similar to Code Example 6.2.

C-Code:

```
int branch_test3(int input) {
  int tmp = 0;
  tmp = (input > 5 &&
         input < 10) ?
         3 : 4;
  tmp = (input > 5) ?
         tmp : -7;
  return tmp*input;
}
```

SPARC V8:

```
 1  branch_test3:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    add %i0, -6, %l0
 5    subcc %l0, 4, %l0
 6    bcc .LBB0_2
 7    nop
 8  ! BB#1:
 9    ba .LBB0_3
10    or %g0, 3, %l0
11  .LBB0_2:
12    or %g0, 4, %l0
13  .LBB0_3:
14    subcc %i0, 5, %l1
15    bg .LBB0_5
16    nop
17  ! BB#4:
18    or %g0, -7, %l0
19  .LBB0_5:
20    smul %l0, %i0, %i0
21    jmp %i7+8
22    restore %g0, %g0, %g0
```

SPARC V8 + `movcc`:

```
 1  branch_test3:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    or %g0, 3, %l0
 5    or %g0, 4, %l1
 6    add %i0, -6, %l2
 7    subcc %l2, 4, %l2
 8    mov[cs] %l0, %l1
 9    or %g0, -7, %l0
10    subcc %i0, 5, %l2
11    mov[g] %l1, %l0
12    smul %l0, %i0, %i0
13    jmp %i7+8
14    restore %g0, %g0, %g0
```

SPARC V8 + `selcc`:

```
 1  branch_test3:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    add %i0, -6, %l0
 5    subcc %l0, 4, %l0
 6    sel[cs] 3, 4, %l0
 7    subcc %i0, 5, %l1
 8    sel[g] %l0, -7, %l0
 9    smul %l0, %i0, %i0
10    jmp %i7+8
11    restore %g0, %g0, %g0
```

and finally set to -7. The corresponding execution path includes basic blocks $0, 2, 3, 4, 5$. For value range (2), `tmp` is set to 3 in the first step, but will not be changed anymore. Hence, this corresponds to executing blocks $0, 1, 3, 5$. For the last case, `tmp` will initially be set to 4 and will not be changed anymore as for the previous case. The according execution path covers basic blocks $0, 2, 3, 5$. Having a look at all three cases reveals that the worst case is reached for conditions (1) and (2). The resulting cycle count is $(5{\cdot}1){+}(1{\cdot}1){+}(3{\cdot}1){+}(1{\cdot}1){+}(1{\cdot}5{+}2{\cdot}1) = 17$ or $(5 \cdot 1) + (2 \cdot 1) + (3 \cdot 1) + (1 \cdot 5 + 2 \cdot 1) = 17$.

Analyzing the assembler code including conditional move instructions is far simpler: The code size is 48 bytes; the single-path solution always takes 16 cycles. The solution with conditional select instructions is even better concerning performance and code size: it only has 36 bytes and takes 13 cycles, which are the lowest numbers of all presented assembler codes. Thus, the current implementation of the LLVM code generating passes needs additional support by the programmer if the resulting code shall be performant, small in size and easy to analyze.

6.1.3 The Impact of Hardware Loops

Code Example 6.4 shows how a bounded C loop is identified by the loop generating pass and correctly transformed to a hardware supported loop. Although the resulting code has one additional block in comparison with the SPARC V8 solution, the analysis is easier: Basic block 0 will be executed only once and takes 3 cycles. The loop initialization is done in block 1, takes 4 cycles and will also be executed only once. All instructions of block 2 take one cycle except for the `ld` instruction, which takes 2 cycles. The resulting $6 \cdot 1 + 1 \cdot 2 = 8$ cycles will be executed 20 times, thus taking 160 cycles. Adding the last two cycles of the instructions in block 3 results in $3 + 4 + 160 + 2 = 169$ cyles.

Calculating the cycle count of the SPARC V8 assembler code of Code Example 6.4 needs additional information: If the analysis tool is able to refer to the original C code, it can easily identify the loop bound for basic block 1. Otherwise, the branch condition for line 15 has to be taken into account. The condition *not equal* is fulfilled if register `%l0` is not equal -20. `%l0` is used as destination register only in lines 4 and 13: The first time it is set to 0, the second time it is decremented. Hence, the branch will be taken 20 times before basic block 2 can be reached. Basic block 0 is executed only once, thus taking exactly 3 cycles. Basic block 1 takes $9 \cdot 1 + 1 \cdot 2 = 11$ cycles and is executed 20 times resulting in 220 cycles. The last block only takes 2 cycles, meaning that the total cycle count is $3 + 220 + 2 = 225$ which is about 33 % more than the solution with hardware loops. The code size of the SPARC V8 is only 60 bytes, whereas the solution including hardware loops needs 64 bytes. Thus, for the current example, there is no solution outperforming the other one in all aspects. Nevertheless, in hard real-time systems, where an easy WCET analysis and performance can be seen more important than code size, the target supporting hardware loops should be preferred.

Code Example 6.4 Implementation of a simple function calculating the sum of an array. As the loop does not contain any branches, the compiler can easily identify the index variable and the loop bound. Thus, the generated code benefits from the hardware loops instruction set extension as can be seen in the lower right example. Although the output for the SPARC V8 ISA is very similar, the calculated cycle count is greater and the WCET analysis might be more complex.

C-Code:

```
#define ARRAY_SIZE 20

int array[ARRAY_SIZE] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20
};

int loop_test(int input) {
  int i, sum = 0;
  for (i = 0; i < ARRAY_SIZE; i++) {
    sum += array[i];
  }
  return sum;
}
```

SPARC V8:

```
 1  loop_test:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    sethi 0, %l0
 5    or %g0, %l0, %i0
 6  .LBB0_1:
 7    sethi %hi(array), %l1
 8    add %l1, %lo(array), %l1
 9    sll %l0, 2, %l2
10    sub %l1, %l2, %l1
11    ld [%l1], %l1
12    add %l1, %i0, %i0
13    add %l0, -1, %l0
14    subcc %l0, -20, %l1
15    bne .LBB0_1
16    nop
17  ! BB#2:
18    jmp %i7+8
19    restore %g0, %g0, %g0
```

SPARC V8 + `hwloop`:

```
 1  loop_test:
 2  ! BB#0:
 3    save %sp, -96, %sp
 4    sethi 0, %l0
 5    or %g0, %l0, %i0
 6  ! BB#1:
 7    hwloop init .LBB0_2, %loops
 8    hwloop init .LBB0_3, %loope
 9    hwloop init 20, %loopb
10    hwloop start
11  .LBB0_2:
12    sethi %hi(array), %l1
13    add %l1, %lo(array), %l1
14    sll %l0, 2, %l2
15    sub %l1, %l2, %l1
16    ld [%l1], %l1
17    add %l1, %i0, %i0
18    add %l0, -1, %l0
19  .LBB0_3:
20    jmp %i7+8
21    restore %g0, %g0, %g0
```

6.1.4 If-then-else Structures and Hardware Loops

When adding a conditional branch to the just presented example, the advantage of introducing multiple instruction set extensions can be verified: In Code Example 6.5, the sum is only calculated if the current index is not equal to 7. As can be seen, the resulting assembler code for the SPARC V8 is more complex and consists of 5 basic blocks and two conditional branches. Unfortunately, LLVM is not able to identify the branch and replace it by a conditional move or select instruction. However, when introducing predicated blocks, the involved basic blocks can be merged and a hardware loop is generated as can be seen in the lower right assembler output of Code Example 6.5.

Calculating the cycle count for the solution including predicated blocks and hardware loops is quite easy because no conditional branches exist: Basic Block 1 and 2 are executed once and take $3 + 4 = 7$ cycles. Basic block 2 is executed 20 times and thus takes $(9 \cdot 1 + 1 \cdot 2) \cdot 20 = 220$ cycles. Adding two cycles of the remaining basic block 3 results in a total cycle count of $7 + 220 + 2 = 229$. The performance evaluation of the SPARC V8 assembler code is a little more complex: From the original C code we know that the loop will be executed 20 times and that basic block 2 will not be executed if the index is equal to 7. Thus, basic blocks 1 and 3 will be executed 20 times, whereas basic block 2 will only be executed 19 times. The resulting cycle count is $3 \cdot 20 = 60$ cycles for basic block 1, $(5 \cdot 1 + 1 \cdot 2) \cdot 19 = 133$ cycles for basic block 2 and $4 \cdot 20 = 80$ cycles for basic block 3. When adding the 5 remaining cycles from basic blocks 0 and 4, the total amount of cycles is $60 + 133 + 80 + 5 = 278$. Note that the introduced `nop` instructions in lines 9 and 21, which are used as delay slots for all types of branches[1], are executed 20 times. Thus, the presented solution spends 40 cycles in `nop` instructions, i.e., doing nothing, whereas the other solution does not have any superfluous instructions. As for Code Example 6.4, the assembler output for the SPARC V8 instruction set has a slightly better code size (72 bytes in contrast to 76 bytes), but has a remarkable greater cycle count, meaning that the solution including the instruction set extensions is more feasible for real-time systems.

6.1.5 Conclusion

Although the presented code examples of the current section might be seen theoretical and cannot cover real-life scenarios, they showed some interesting aspects concerning the proposed instruction set extensions. This leads to the following assumptions:

(1) The manual timing analysis of a given code gets more complex when introducing loops and multiple conditional structures such as if-then-else blocks.
(2) If the resulting assembler code makes use of instruction set extensions, the manual analysis is easier and sometimes even trivial because there is a single-path solution.
(3) Some code examples only benefit from the instruction set extensions if they are combined, e.g., predicated blocks and hardware loops.
(4) For some examples, the generated assembler output with instruction set extensions is easier to analyze and shows better performance than the solution for the original SPARC V8 ISA. However, this is not generally true for the resulting code size.

[1] See Section 4.2.2 for details.

Code Example 6.5 Slightly adapted implementation of Code Example 6.4. The loop now exhibits a conditional branch instruction which is too complex for the compiler to be replaced by a conditional move or conditional select. The hardware loop generating pass in its current implementation only handles loops covering a single basic block. Hence, the assembler output for targets with support for conditional move and hardware loop instructions is equal to the SPARC V8 output. However, the branch elimination pass introducing predicated blocks is able to unite all blocks of the loop. Consequently, the resulting assembler output shows hardware loop instructions as can be seen in the lower right example.

C-Code:

```
#define ARRAY_SIZE 20

int array[ARRAY_SIZE] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20
};

int loop_test(int input) {
  int i, sum = 0;
  for (i = 0; i < ARRAY_SIZE; i++) {
    if (i != 7) {
      sum += array[i];
    }
  }
  return sum;
}
```

SPARC V8:

```
1  loop_test:
2  ! BB#0:
3    save %sp, -96, %sp
4    sethi 0, %l0
5    or %g0, %l0, %i0
6  .LBB0_1:
7    subcc %l0, -7, %l1
8    be .LBB0_3
9    nop
10 ! BB#2:
11   sethi %hi(array), %l1
12   add %l1, %lo(array), %l1
13   sll %l0, 2, %l2
14   sub %l1, %l2, %l1
15   ld [%l1], %l1
16   add %l1, %i0, %i0
17 .LBB0_3:
18   add %l0, -1, %l0
19   subcc %l0, -20, %l1
20   bne .LBB0_1
21   nop
22 ! BB#4:
23   jmp %i7+8
24   restore %g0, %g0, %g0
```

SPARC V8 + `predblock` icc + `hwloop`:

```
1  loop_test:
2  ! BB#0:
3    save %sp, -96, %sp
4    sethi 0, %l0
5    or %g0, %l0, %i0
6  ! BB#1:
7    hwloop init .LBB0_2, %loops
8    hwloop init .LBB0_3, %loope
9    hwloop init 20, %loopb
10   hwloop start
11 .LBB0_2:
12   subcc %l0, -7, %l1
13   predbegin[ne]
14   sethi %hi(array), %l1
15   add %l1, %lo(array), %l1
16   sll %l0, 2, %l2
17   sub %l1, %l2, %l1
18   ld [%l1], %l1
19   add %l1, %i0, %i0
20   predend
21   add %l0, -1, %l0
22 .LBB0_3:
23   jmp %i7+8
24   restore %g0, %g0, %g0
```

6.2 Evaluation of Selected Algorithms

In [Pus07], Peter Puschner analyzes the impacts of different implementations of the same algorithms on the generated assembler code. The target processor is a Motorola M-Core, which provides conditional move and several simple predicated instructions. The assembler code has been simulated and the resulting cycle counts documented. Most of the implementations are written in the C programming language, but there are several solutions that use inline assembler. Thus, these implementations cannot be used in the current section, because the SPARC V8 instruction set and the proposed extensions are completely different to the M-Core processor.[2]

The following subsections show how the coding style in a high level language influences the resulting assembler code. The performance evaluation has been done with an instruction set simulator, implementing the SPARC V8 ISA and the presented extensions. In a first step, the C code is translated to LLVM IR by the `llvm-gcc` with optimization level 3 (option `-O3`). Afterwards, SPARC V8 assembler code is generated by LLVM and the implemented additional passes. It has been tried to find the corresponding worst-case scenarios of every algorithm to draw conclusions about the feasibility of the proposed instruction set extensions. In most cases, a preferred solution could be found. The detailed results of the performance evaluation are listed in Appendix A. The analyzed targets implemented the following instruction set extensions:

- *v8* – The original SPARC V8 instruction set without any extensions.
- *v8-predblockicc* – The original SPARC V8 instruction set with predicated blocks based on integer condition codes.
- *v8-hwloop* – The original SPARC V8 instruction set with additional support for hardware loops.
- *v8-movcc* – The original SPARC V8 instruction set with conditional move instructions and support for hardware loops.
- *v8-selcc* – The original SPARC V8 instruction set with conditional select instructions and support for hardware loops.

6.2.1 Bubble Sort

Bubble Sort Version 1

Bubble sort was the first algorithm which was analyzed in [Pus07]. The standard implementation, denoted *version 1* in the paper, is shown in Code Example 6.6. As can easily be seen, the worst-case scenario occurs when the if-then block is executed for every loop iteration. This is the case if the values of the input array are given in reverse order. Similar to Code Example 6.2, the LLVM compiler is not able to introduce conditional move or select instructions, but can eliminate the most inner branch when predicated blocks are enabled. Although the number of basic blocks is reduced by four in comparison to the SPARC V8 solution, the code size (368 bytes) and the worst-case cycle count (99894 cycles for an array with 100 elements) are equal. However,

[2] The original C code of all described algorithms of the current section are available on the repository of the simulator (`https://github.com/cgeyer/Sparc-V8-IS-extension-simulator/`) in the `examples/cfiles` folder.

the generated code for the *v8-predblockicc* target is a single-path solution, meaning that the best, the average and the worst-case performance are identical.

Code Example 6.6 Traditional implementation of the *bubble sort* algorithm. See [Pus07, p. 7].

```
static void BubbleSort(int a[]) {
  int i, j, t;
  for(i=SIZE-1; i>0; i--) {
    for(j=1; j<=i; j++) {
      if (a[j-1] > a[j]) {
        t = a[j];
        /* swap */
        a[j] = a[j-1];
        a[j-1] = t;
      }
    }
  }
}
```

Bubble Sort Version 2

Version 2 of the *bubble sort* algorithm is shown in Code Example 6.7 and is a single-path implementation. Now, the LLVM compiler is able to recognize CFG patterns to insert conditional move and select instructions. Of course, the assembler output for the *v8* and the *v8-hwloop* targets are identical, because the inner loop consists of too many branches if neither conditional move or select instructions, nor predicated blocks are supported. The resulting performance decreases dramatically, meaning that the best case cycle count is 105042 for the current implementation, which is more than the worst-case performance of the traditional solution. Moreover, the code size and the number of basic blocks increases for these targets.

However, analyzing the resulting assembler output of the *v8-movcc* and *v8-selcc* targets reveals that the single-path solution is quite feasible: Although the code size increases (384 bytes for conditional move, 380 bytes for conditional select), the number of basic blocks and conditional branches slightly decreases. Moreover, it is a single-path solution for both targets with a constant cycle count of 87420 cycles (*v8-movcc*) and 82371 cycles (*v8-selcc*). This is about 20 % less than the best-case (!) cycle count of the same algorithm on the *v8* target and about 15 % less than the worst-case cycle count of the traditional implementation.

Bubble Sort Version 4 & 5

The third version of *bubble sort* was implemented in assembler by Puschner and therefore is not part of the current evaluation. Version 4 is based on the traditional implementation, but introduces a `break` instruction causing an early exit from the outer loop in case the list has already been sorted. On the one hand, this *optimization* increases best and average case performance (the best case takes 1398 cycles), on the other hand, it decreases the worst-case performance (104946 cycles). Version 5 is very similar, but the exit condition is in the loop header, such that no `break` instruction was used. Nevertheless, the resulting assembler code is nearly the same

Code Example 6.7 Single-path implementation of the *bubble sort* algorithm. See [Pus07, p. 8].

```
static void BubbleSort(int a[]) {
  int i, j, s, t;
  for(i=SIZE-1; i>0; i--) {
    for(j=1; j<=i; j++) {
      s = a[j-1];
      t = a[j];
      a[j-1] = ((s <= t) ? s : t);
      /* a[j-1] = min(s, t); */
      a[j] = ((s > t) ? s : t);
      /* a[j] = max(s, t); */
    }
  }
}
```

and shows no differences in performance. Hence, both solutions are not feasible for real-time systems.

Bubble Sort Version 6

The last evaluated implementation of the *bubble sort* algorithm is based on version 5. The main difference is that the outer loop is not exited earlier. To avoid that the instructions of the inner loop are still executed, the swapping is conditionally executed as can be seen in Code Example 6.8. Although the resulting assembler code is a single-path solution, the second version of *bubble sort* shows better performance: The code size is 440 bytes in contrast to 384 bytes for the *v8-movcc* target and the worst-case performance is about 5000 cycles worse.

Code Example 6.8 Second version of a single-path implementation of the *bubble sort* algorithm. See [Pus07, p. 16].

```
static void BubbleSort(int a[]) {
  int i, j, s, t;
  int exchanged=1;
  int loop_is_over;
  for(i=SIZE-1; i>0; i--) {
    loop_is_over = (exchanged == 0);
    exchanged = 0;
    for(j=1; j<=i; j++) {
      s = a[j-1];
      t = a[j];
      a[j-1] = (loop_is_over || (s <= t)) ? s : t;
      a[j] = (loop_is_over || (s <= t)) ? t : s;
      exchanged = (loop_is_over || (s <= t)) ? exchanged : 1;
    }
  }
}
```

Summary

The evaluation results in some parts correspond to the results by Puschner, but there are some remarkable differences: The gap between best case and worst-case performance is much higher for the presented benchmarks of the current thesis than it was in [Pus07]. Moreover, the second version shows the best worst-case performance in case the target supports conditional move or conditional select instructions. In the paper, the best worst-case performance was achieved by the first version, disregarding the solutions implemented in assembler.

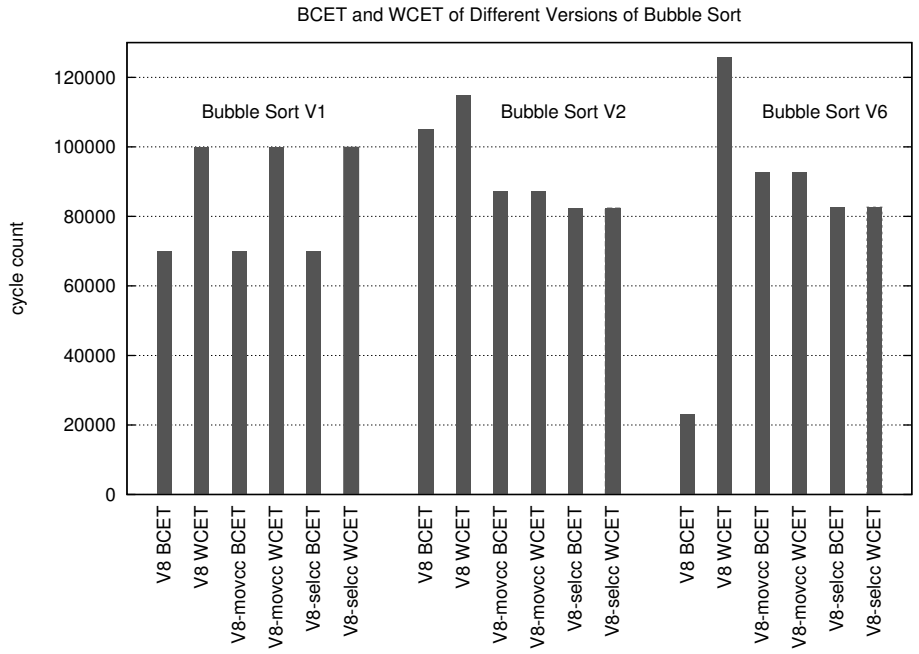

Figure 6.1: BCET and WCET of different implementations of *bubble sort* simulated on several SPARC V8 targets.

Figure 6.1 summarizes the results of the current performance evaluation of *bubble sort*: As can be seen easily, the cycle count of the first version is equal for the *v8*, *v8-movcc* and *v8-selcc* target. However, in version 2, the *v8-movcc* and *v8-selcc* targets show constant execution time, which is even better than the worst-case cycle count of the first version. Version 6 shows a much greater worst-case cycle count on the the *v8* target, but has a better worst-case performance on the *v8-movcc* and *v8-selcc* targets than the original solution. Summarizing, introducing conditional move or select instructions in combination with hardware loops improves the worst-case performance and the control flow analysis of *bubble sort* if the programmer finds a feasible single-path implementation. The detailed best- and worst-case cycle counts as well as the code size of each implementation for every target can be found in Tables A.1 and A.2 in Appendix A.1.

6.2.2 Find First

The problem statement for the *find first* algorithm is quite simple: The first occurrence of a given value within in a specified input array of arbitrary integers shall be identified. The key does not need to be part of the array. The algorithm shall either return the index of the array element or an error code in case the key cannot be found. The test input vectors for all versions of the algorithm were taken from the code listing on page 62f in [Pus07]. They include single, multiple and no occurrences of the key in the array. The latter is the worst-case scenario for most implementations.

Find First Versions 1 & 2

A naive implementation of *find first* is shown in Code Example 6.9. When translating this version, LLVM is not able to use any of the proposed instruction set extensions such that the resulting code size (152 bytes) and the worst-case cycle count (107 cycles) are equal for all targets. The second version does not use any `break` instruction, but introduces a boolean variable which is set when the key is found in the array. In every loop iteration, it has to be checked whether the index variable is smaller than the array size and if the key has already been found. This overhead needs 28 additional bytes for the SPARC V8 target, but only 8 bytes more when using conditional selects. Unfortunately, the resulting worst-case performance is lower (172 cycles for the SPARC V8 target, 132 cycles for the *v8-selcc* target) than the first implementation. Only the best case execution time could be improved by one cycle for the targets supporting conditional moves and selects.

Code Example 6.9 Traditional implementation of the *find first* algorithm. See [Pus07, p. 21].

```
static int findfirst(int key, int a[]) {
  int i;
  int position = SIZE;
  for(i=0; i<=SIZE-1; i++) {
    if (a[i] == key) {
      position = i;
      break;
    }
  }
  return position;
}
```

Find First Version 3

The third version makes use of the ternary `?`-operator to get a single-path transformation of the second implementation as can be seen in Code Example 6.10. When the compiler settings include loop unrolling, the resulting code sizes for all targets are extraordinary large: The smallest assembler code is generated for the *v8-selcc* target (484 bytes), the largest for the *v8-predblockicc* target (892 bytes). The code size is only slightly larger than that of the first version when turning off loop unrolling optimizations.

Regarding worst-case performance, the assembler output of the third version with loop unrolling has a smaller cycle count for the *v8-selcc* target in comparison with the first version on all targets. When disabling loop unrolling, the cycle count increases for all targets, making this solution very unattractive. Thus, version 3 is only feasible for real-time systems which do not rely on small instruction memories.

Code Example 6.10 Single-path implementation of the *find first* algorithm. See [Pus07, p. 23].

```
static int findfirst(int key, int a[]) {
  int i;
  int position = SIZE;
  int found = 0;
  int cond;
  for(i=0; i<=SIZE-1; i++) {
    cond = !found;
    cond = cond & (a[i] == key);
    found = (cond ? 1 : found);
    position = (cond ? i : position);
  }
  return position;
}
```

Find First Version 5 & 6

The fifth version of *find first*, which is shown in Code Example 6.11, uses a completely different approach: To avoid that the index is overwritten if the key occurs more than once, the search starts from the end of the array and goes to its beginning. Thus, only the first occurrence is saved in the `position` variable. In version 6, the inner if-then block is replaced by a `?`-operator, but there is no difference in the generated assembler outputs. Although the resulting code sizes are greater than the first implementation, the worst-case cycle count could be decreased to 40 cycles for the *v8-selcc* target. However, when disabling the loop unrolling optimization as for version 3, the worst-case cycle count is 141 cycles for targets with conditional move or select instructions.

Code Example 6.11 Simple backward loop implementation of the *find first* algorithm. See [Pus07, p. 26].

```
static int findfirst(int key, int a[]) {
  int i;
  int position = SIZE;
  for(i=SIZE-1; i>=0; i--) {
    if (a[i] == key) {
      position = i;
    }
  }
  return position;
}
```

Summary

The resulting code sizes and cycle counts of all versions are listed in Tables A.3 and A.4. In contrast to *bubble sort*, there does not exist an ideal solution: From the performance point of view, the fifth version seems to be preferable when conditional select instructions are available (version 5/6a). Nevertheless, loop unrolling is only suitable for arrays with few elements, because the code size increases with every unrolled loop iteration. Moreover, when dealing with arrays of variable size, it is not possible to use this kind of optimization. Figure 6.2 shows the best- and worst-case cycle counts of all versions with disabled loop unrolling: As can be seen, there is no solution which has a better worst-case performance than the first implementation on any target even if conditional move or select instructions and hardware loops are enabled. However, version 3 with no loop unrolling (version 3b) seems to be the most adequate and flexible solution for targets supporting conditional select instructions: The algorithm has constant execution time and is about 18 % slower in the worst-case than the original solution; its code size is 20 bytes larger.

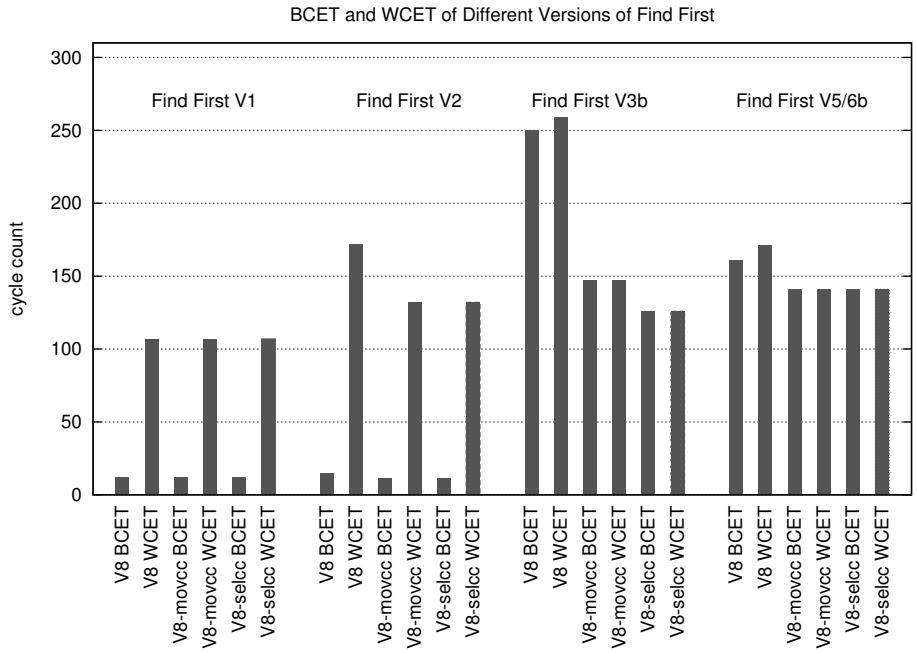

Figure 6.2: BCET and WCET of different implementations of *find first* simulated on several SPARC V8 targets. The *b* after the version number denotes that loop unrolling has been disabled by the compiler settings for the evaluation.

6.2.3 Binary Search

In contrast to *find first*, *binary search* tries to find a given value within a presorted array. In [Pus07], there are 11 different implementations of this algorithm; 7 are based on C code. The current section will only go into details for three of them. The loop unrolling optimization has been generally disabled, such that the resulting code sizes and cycle counts do less depend on compiler optimizations. The used input vector was a logarithmically distributed array of 16 integers. The covered test scenarios searched for each array element. It has also been tried to find the worst-case scenario by searching for one value which is greater than the last array element.

Binary Search Version 1

The traditional implementation can be seen in Code Example 6.12: As soon as the key is found, the function returns the corresponding index. Thus, the best case scenario occurs if the key is saved in the center element of the array. The generated assembler output for the SPARC V8 instruction set has a code size of 208 bytes, a worst-case performance of 101 cycles and a best case cycle count of just 10 cycles. LLVM is able to eliminate branches such that the code size and worst-case execution time on targets with conditional move or select instruction is slightly reduced.

Code Example 6.12 Traditional implementation of the *binary search* algorithm. See [Pus07, p. 30].

```
static int binSearch(int key, int a[]) {
  int left = 0, right = SIZE - 1, idx, inc;
  do {
    idx = (right + left) >> 1;
    if (a[idx] == key) {
      return idx;
    }
    else if (a[idx] < key) {
      left = idx+1;
    }
    else {
      right = idx-1;
    }
  } while (right >= left);
  return -1;
}
```

Binary Search Version 2 & 3

The second version introduces a boolean variable, which saves whether the key has already been found. The loop is exited if the condition is true. However, the resulting assembler code is identical to the first version on all targets. Version 3 is a first trial of a single-path implementation,

but the C code is too complex for the compiler such that the resulting assembler output shows worse performance and a greater code size in comparison with the first two implementations.

Binary Search Version 5

The first successful single-path transformation is achieved in version 5: The resulting code size is only 5 to 10 % more than the original variant and the worst-case performance is better on all targets, including the SPARC V8 without any instruction set extensions. Nevertheless, only targets on which branches could be removed, e.g., by enabling predicated blocks or conditional move instructions, show constant execution times for this implementation.

Binary Search Version 7

Code Example 6.13 presents a solution which does not seem to be a single-path implementation at the first glance, but can be translated by the compiler to achieve constant execution times for all tested scenarios. In contrast to all preceding variants, the left search boundary variable saves the index, which will be finally returned. Except for the *v8-selcc* target, the resulting solution shows the best worst-case performance so far[3] and has nearly the same code size as the first version.

Code Example 6.13 Single-path implementation of the *binary search* algorithm. See [Pus07, p. 43].

```
static int binSearch(int key, int a[]) {
  int left = 0, right = SIZE - 1, idx, inc;
  do {
    idx = (left + right) >> 1;
    if (key > a[idx]) {
      left = idx + 1;
    } else {
      right = idx;
    }
  } while (left < right);
  if (a[left] < key) {
    return -1;
  } else {
    return left;
  }
}
```

Binary Search Version 8 & 10

Version 8 is a modification of version 7 and makes use of the ternary `?`-operator. Like the third version, the C code is too complex for the compiler and the resulting assembler code is no

[3] The reason for this is a bug in the code generating pass for hardware loops, which inserts a loop where it should not be done. When disabling this pass, the performance is about the same as on the *v8-movcc* target.

single-path solution. Code Example 6.14 (version 10) is an improved variant of version 8 for which the LLVM compiler is able to identify the loop bound (i.e., $\log_2 16 = 4$) automatically, such that a correct hardware loop may be introduced. The best performance is achieved on the *v8-selcc* target, which has a constant cycle count of only 57 cycles. In comparison to the first implementation on the *v8* target, this is an improvement by nearly 44 % and the code size has even decreased by 8 bytes.

Code Example 6.14 Slightly improved single-path implementation of the *binary search* algorithm. See [Pus07, p. 49].

```
static int binSearch(int key, int a[]) {
  int left = 0, right = SIZE - 1, idx, inc;
  for (inc = SIZE-1; inc > 0; inc = inc >> 1) {
    idx = (left + right) >> 1;
    left = (key > a[idx]) ? idx + 1 : left;
    right = (key <= a[idx]) ? idx : right;
  }
  return (a[left] < key) ? -1 : left;
}
```

Summary

The complete evaluation of all versions on all targets can be found in Tables A.5 and A.6. As can be seen in Figure 6.3, the first implementation of *binary search* shows an excellent best-case performance on many targets, but has no competitive worst-case cycle count. The best results may be achieved with version 10 on targets with conditional move or select instructions and hardware loops: The resulting assembler code shows constant execution time and the worst-case cycle count is about 43 % less than the original worst-case cycle count on the SPARC V8 target with no instruction set extensions.

6.2.4 Increment Multi-byte Counter

The fourth algorithm which was analyzed by Puschner, implements a simple counter which is not restricted to a fixed bit size. The value is saved in an array of 8-bit integers, such that a counter with an arbitrary bit width may be used. The implemented test scenario increments an 80-bit counter from the initial maximum value, which is $2^{80} - 1$, to $65\,535$. Note that an integer overflow occurs for the the first incremental step from the maximum counter value to zero such that all bytes of the counter array have to be incremented. Although it does not cover all possible test cases, it can be verified whether the resulting assembler code shows constant performance on a wide range of input values. As for *binary search*, the loop unrolling optimization has been generally disabled.

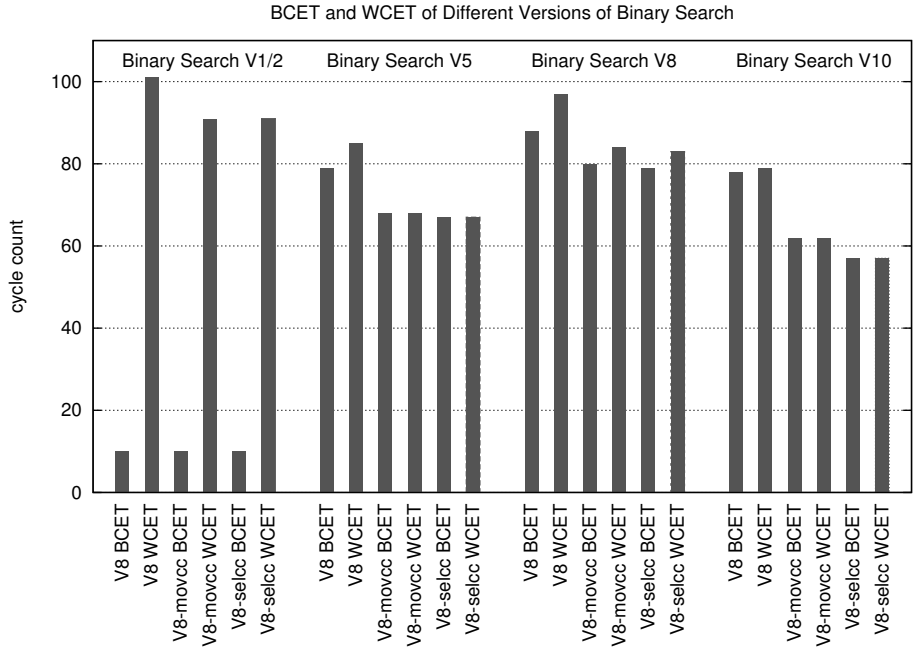

Figure 6.3: BCET and WCET of different implementations of *binary search* simulated on several SPARC V8 targets.

Multi-byte Counter Version 1

Code Example 6.15 shows a first simple implementation of the *multi-byte counter*: Per default, the first byte of the counter is incremented. If the resulting value equals 0, the next byte is incremented, otherwise, the loop is exited. The resulting assembler code is identical for all targets and has a good best case performance of only 16 cycles. The worst case, which occurs if all bytes have to be incremented, takes 142 cycles.

Code Example 6.15 Traditional implementation of *incrementing a multi-byte counter*. See [Pus07, p. 54].

```
static void inc_counter(COUNTER counter) {
  int idx;
  for(idx=0; idx<COUNTERSIZE; idx++) {
    unsigned char tmp;
    tmp = counter[idx];
    counter[idx] = tmp+1;
    if (counter[idx] > 0)
      break;
  }
}
```

Multi-byte Counter Version 2 & 4

The second version, which is shown in Code Example 6.16, is a single-path implementation of the first version. Due to its increased complexity, the code size is larger for all evaluated targets. Moreover, the worst-case cycle count has increased, except for the target supporting conditional selects for which it is about the same as for the original implementation. Code Example 6.17 is an improved single-path variant based on the following idea: Each byte is incremented by a certain value, which is saved in a separate variable. Initially, the value is set to 1, but it gets cleared as soon as no byte overflow occurs. The resulting assembler output has nearly the same size as the first version and shows a better worst-case performance on targets with conditional move or select instructions.

Code Example 6.16 First single-path implementation of *incrementing a multi-byte counter*. See [Pus07, p. 55].

```
static void inc_counter(COUNTER counter) {
  int idx;
  int loop_is_over = 0;
  for(idx=0; idx<COUNTERSIZE; idx++) {
    unsigned char tmp;
    tmp = counter[idx];
    counter[idx] = loop_is_over ? counter[idx] : tmp+1;
    loop_is_over = loop_is_over || (counter[idx] > 0);
  }
}
```

Code Example 6.17 Improved single-path implementation of *incrementing a multi-byte counter*. See [Pus07, p. 57].

```
static void inc_counter(COUNTER counter) {
  int idx, inc_val;
  inc_val = 1;
  for(idx=0; idx<COUNTERSIZE; idx++) {
    unsigned char tmp;
    tmp = counter[idx];
    counter[idx] = tmp + inc_val;
    if (counter[idx] > 0)
      inc_val = 0;
  }
}
```

Summary

Tables A.7 and A.8 give a complete overview of all resulting code sizes and cycle counts. Figure 6.4 outlines the best- and worst-case performance of different implementations of the algorithm simulated on several targets: Version 2 is a single-path solution in case of targets with

conditional move or select instructions, but the worst-case cycle count is slightly larger than that of the original implementation. For Version 4, there is nearly no difference between best- and worst-case cycle count on the *v8* target, meaning that even on a target with no support of branch elimination, a quasi single-path solution might be found. Moreover, on the *v8-movcc* and *v8-selcc* targets, a much lower worst-case cycle count than that of the original solution can be achieved.

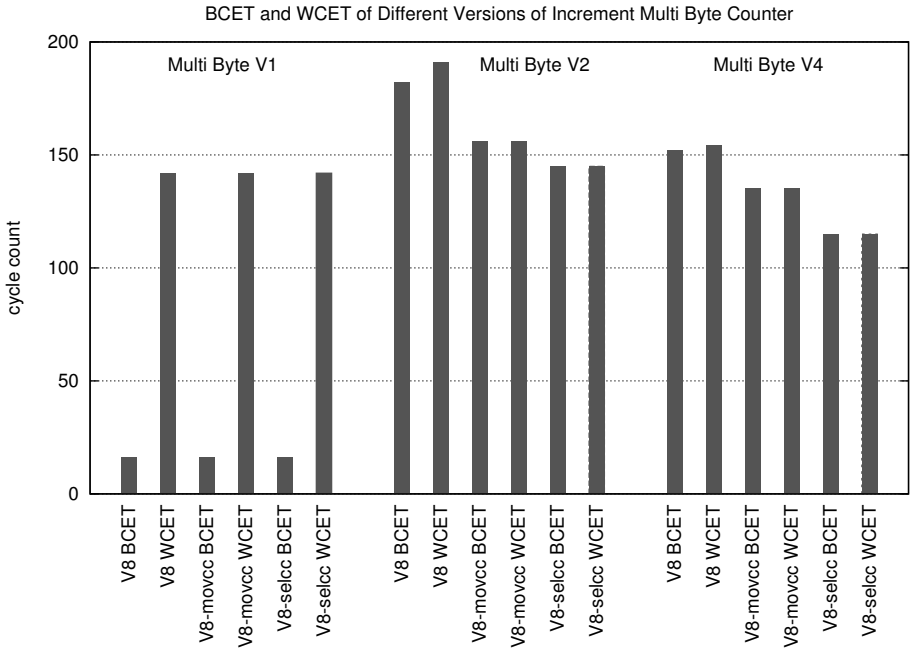

Figure 6.4: BCET and WCET of different implementations of *multi-byte counter* simulated on several SPARC V8 targets.

6.2.5 Concluding Remarks

As could be seen, enabling several of the instruction set extensions and changing the implementation of well-known algorithm can be very suitable in real-time environments: Three out of four algorithms could be transformed such that the resulting assembler code is a single-path solution and shows a lower worst-case cycle count than the first implementation without the additional instructions. Nevertheless, one has to take several aspects into account: The used LLVM compiler is not optimized for generating assembler output for real-time systems, i.e., single-path codes or small code sizes. Moreover, the implemented code generating passes for hardware loops and branch elimination are not perfect and may be still improved. However, the LLVM IR language has a conditional select instruction, meaning that the generated assembler output for targets with hardware support for similar instructions has a better performance than targets where they have

to be emulated. Thus, the presented results are only valid for the current evaluation settings and may not be feasible for other compilers and processors.

6.3 Towards a Time Predictable Instruction Set

6.3.1 Simulation Setup and Problem Statement

The previous sections showed the impacts of different coding styles on the generated assembler output and identified useful instruction set extensions in terms of code size, number of branches and worst-case behavior. However, the results were restricted to a small amount of algorithms executed only on a few targets. Thus, the current section tries to find the most suitable combination of instruction set extensions such that

(1) the resulting assembler code is easy to analyze,
(2) the worst-case performance is still competitive, and
(3) the code size of the program does not increase dramatically.

In order to do so, 24 algorithms have been compiled for 13 different combinations of the SPARC V8 instruction set including the additional instructions. A short overview of the evaluated targets is given in Section 6.3.2; the algorithms are shortly described in Section 6.3.3. The resulting assembler code has been simulated and the following key figures recorded:

- *Code Size* is the code size of the complete program, including test and main functions, in bytes. Although memory size is usually no issue for desktop applications, it can be an important factor for embedded systems.
- *Conditional Branches* denotes the number of conditional branches within the complete program. It is an indication of the complexity of the control flow graph. A lower number means that the complexity of the control-flow graph decreases and the worst-case path is easier to identify.
- *Unconditional Branches* denotes the number of (unconditional) jump instructions within the complete program. They usually do not have an influence on the complexity of the control flow graph.
- *Number of NOPs* denotes the number of delay instructions which have to be inserted after a conditional or unconditional branch, see Section 4.2.2 for details. It is an indicator for the number of superfluous instructions, i.e., instructions doing nothing.
- *Number of MBBs* is the amount of machine basic blocks of the whole program and corresponds to the number of vertices of the control flow graph. Like the for conditional branches, a lower number is an indicator that the worst-case path of the given assembler code is easier to identify.
- *Min/Max Cycles* is the measured minimum/maximum cycle count. It does not necessarily need to be the actual worst-case execution time. For algorithms with a single vector of input values, only the measured cycle count has been recorded.
- *Deviation* is calculated by the formula specified in Equation 6.1 and is only stated if the minimum and maximum cycle count of the current algorithm are available.

- *Number of movCCs* denotes the number of conditional move instructions which have been used within the whole program.
- *Number of selCCs* denotes the number of conditional select instructions which have been used within the whole program.
- *Number of predbegins* is equivalent to the number of basic blocks which have been predicated.
- *Nesting level of predicated blocks* is only available for predicated instructions based on predicate registers. It is equivalent to the highest predicate register which has been used within the whole program. This relates to the nesting level of if-then-else structures which could be eliminated.
- *Number of HWLoops* denotes the number of used hardware loops within the whole program.

In [TW04, p. 160], it is stated that *the differences between the upper and lower bounds and the worst and best cases of execution times, respectively, are measures for the timing predictability of the whole system.* In other words, it can be said that hardware and software behave predictably if the best- and worst-case execution times are either equal or nearly the same. A system has a low *timing predictability* if the measured minimum and maximum cycle counts of the same algorithm are highly divergent. Based on these definitions, the *deviation* key figure used in the current evaluation has been introduced in order to provide an indicator of *timing predictability*:

$$d = \frac{t_{\max} - t_{\min}}{t_{\max}}, \quad d \in [0, 1) \tag{6.1}$$

The difference between measured minimum and maximum cycle count is divided by the maximum cycle count. The result d is a real number between zero and one. If the deviation is 0, the algorithm has constant execution time, i.e., the minimum and maximum execution times are equal; a value near 1 indicates a large deviation. If a target shows a deviation of 0 for a number of algorithms, it can be said to be provide *timing predictability* and might thus be suitable for real-time environments. In [KP11], Kirner and Puschner give a formula for the *stability* of a system, which is quite similar to the just presented definition of the *deviation*, see Equation 6.2. The main difference is that a system may be called *stable* if the *stability* is near or equal 1, which is in fact the opposite of the definition of the *deviation*.

$$stability(tm) = \frac{BCET_{tm}}{WCET_{tm}} \tag{6.2}$$

Another key figure for the current evaluation is the number of occurrences of each of the instruction set extensions: On the one hand, it shows which algorithms may benefit (more) from the newly introduced instructions. On the other hand, it indicates whether an additional instruction is only useful in combination with another: If, for example, multiple hardware loops are only introduced if predicated blocks have been enabled, it does not make any sense to add support for hardware loops without predicated blocks.

6.3.2 Evaluated Combinations of Instruction Set Extensions

For the current evaluation, all algorithms have been translated to assembler code and simulated on several targets covering nearly all possible combinations of the presented instruction set extensions. The following 13 targets have been defined:

- *v8* – The original SPARC V8 instruction set without any extensions.
- *v8-m* – The original SPARC V8 instruction set with conditional move instructions.
- *v8-s* – The original SPARC V8 instruction set with conditional select instructions.
- *v8-i* – The original SPARC V8 instruction set with predicated blocks based on integer condition codes.
- *v8-r* – The original SPARC V8 instruction set with predicated blocks based on predicate registers.
- *v8-l* – The original SPARC V8 instruction set with hardware loops.
- *v8-ml* – The original SPARC V8 instruction set with conditional move instructions and hardware loops.
- *v8-sl* – The original SPARC V8 instruction set with conditional select instructions and hardware loops.
- *v8-il* – The original SPARC V8 instruction set with predicated blocks based on integer condition codes and hardware loops.
- *v8-rl* – The original SPARC V8 instruction set with predicated blocks based on predicate registers and hardware loops.
- *v8-mil* – The original SPARC V8 instruction set with conditional move instructions, predicated blocks based on integer condition codes and hardware loops.
- *v8-srl* – The original SPARC V8 instruction set with conditional select instructions, predicated blocks based on predicate registers and hardware loops.
- *v8-sil* – The original SPARC V8 instruction set with conditional select instructions, predicated blocks based on integer condition codes and hardware loops.

6.3.3 Evaluated Algorithms

The current section will only give a short overview of the evaluated algorithms. A short listing of them as well as the key figures for each target are given in Appendix B. Moreover, the complete simulation environment is available on GitHub (`https://github.com/cgeyer/Sparc-V8-IS-extension-simulator/`) in the `benchmarks` folder.

One part of the benchmarks algorithms has been taken from [Mal11], the WCET project website of the Swedish Mälardalen university. The majority of algorithms has been implemented by students of TU Vienna participating in a lecture held by Peter Puschner. They were collected and maintained by Benedikt Huber. Most of them are available in multiple versions: one traditional implementation and one or more single-path solutions. The remaining benchmark algorithms were implemented by myself and are also available in a traditional and a single-path transformed variant.

- *Binary Greatest Common Divisor (bgcd):* This implementation of the *greatest common divisor* algorithm by Euclid does neither use modulo nor division operations. All cal-

culations are based on addition, subtraction and shift operations, which are part of most instruction sets.

- *Binary Greatest Common Divisor – Single-Path (bgcd_sp):* This is the single-path implementation of the binary version of the Euclid algorithm.
- *Binary Search (bs):* The first implementation of the *binary search* algorithm is similar to the traditional implementation presented in Section 6.2.3.
- *Binary Search – Single-Path for Fixed Size (bs_sp):* This first single-path implementation of the *binary search* algorithm is equivalent to version 5 in [Pus07].
- *Binary Search – Single-Path for Variable Size (bs_wcet):* This is an improved version of the first single-path implementation of *binary search*.
- *Bubble Sort – Worst Case Scenario (bubble_sort):* This is a performance test for the worst-case scenario of the *bubble sort* algorithm.
- *Switch Case Test (cover):* This test implements three functions dealing with a loop and a tremendous amount of switch-case instructions.
- *Dijkstra Algorithm (dijkstra):* The *Dijkstra algorithm* identifies the shortest path within a given graph.
- *Dijkstra Algorithm – Single-Path (dijkstra_sp):* First single-path implementation of the *Dijkstra algorithm*.
- *Dijkstra Algorithm – Optimized Single-Path (dijkstra_wcet):* Optimized single-path implementation of the *Dijkstra algorithm*.
- *Fourier Discrete Cosine Transformation (fdct):* This algorithm implements the *Fourier discrete cosine transformation* of a 8×8 matrix.
- *Interpolation Table (interpolate):* This algorithm implements the calculation of a mathematical function of which only a few values are known. The output value is calculated by interpolation between the two nearest available values.
- *Interpolation Table – Single-Path (interpolate_sp):* Single-path implementation of the *interpolation table* algorithm.
- *Matrix Sum (matrixsum):* This algorithm calculates the sums of every row and every column of a given matrix and is thus mainly dependent on arithmetic instructions.
- *Median with Quick Sort (median):* The first version of finding the median element of a given array uses quick sort and returns the middle element of the sorted array.
- *Median without Sorting (median_torben):* The second version of calculating the median value is based on the idea to guess the median in a first step and count the number of elements which are greater and which are less. If both values are less or equal to half of the number of array elements, the median has been found, otherwise, a new maximum or minimum value for the next guess is set.
- *Median without Sorting – Single-Path (median_sp):* Single-path implementation of the previously described algorithm.
- *Shell Sort (shellsort):* The *shell sort* algorithm is based on insertion sort, but divides the array into small parts in a first step.[4]
- *Shell Sort – Single-Path (shellsort_sp):* Single-path implementation of the *shell sort* algorithm.

[4]For a detailed description see `http://en.wikipedia.org/wiki/Shellsort`, accessed 2012-02-10.

- *Software Division – Naive Implementation (swdivision):* The first version of the *software division* algorithm simply subtracts the divisor from the dividend, saves the result to the dividend and increments a counter. This is done as long as the dividend is greater than the divisor. The counter value is the result of the division.
- *Software Division – Shift Implementation (swdivision_shift):* The second variant of the *software division* algorithm is based on the manual division as usually learned in school for decimal numbers: It is based on shifting the divisor to the left and subtract it from the dividend.
- *Software Division – Single-Path (swdivision_sp):* Single-path implementation of the *software division* algorithm based on shifting.
- *Threshold (threshold):* The *threshold* algorithm checks for every element of an array, whether it is within the given bounds, i.e., a maximum and a minimum level.
- *Threshold – Single-Path (threshold_sp):* Single-path implementation of the previously described *threshold* algorithm.

6.3.4 Results

Arithmetic Benchmarks

A first glance at the evaluation results reveals that all instruction set extensions cannot improve the performance or reduce the number of branches in scenarios which involve complex arithmetic operations: The *Fourier discrete cosine transformation* introduces hardware loops and conditional move and select instructions, but the resulting performance is nearly the same as for the original SPARC V8 instruction set. In case of calculating the sums of each row and each column of a given matrix (*matrix sum*), LLVM is not able to introduce any of the proposed instruction set extensions such that the resulting assembler code is equal for all targets. For these two algorithms, additional SIMD instructions could have provoked an increase in performance.

Bubble Sort and Switch Case Test

Having a look at the performance evaluation of *bubble sort* shows that nearly no instruction set extension could be introduced by LLVM. The evaluated implementation is similar to the first version of *bubble sort* presented in Section 6.2.1. It has already been shown that other versions are more effective such that the results of this benchmark are not fully representative. Another algorithm which only sparely benefits from the instruction set extensions is the *switch case test*: It just consists of a huge look-up table, which introduces a lot of branches, but cannot be optimized by the compiler. Hence, this benchmark is more interesting to test the performance of a WCET analysis tool or of a code generator and not that of a processor.

Binary Greatest Common Divisor

Nearly all single-path implementations of a given algorithm could benefit from the instruction set extensions: The *binary greatest common divisor* only has 3 conditional branches and only about 50 % of the worst-case cycle count when comparing a target with hardware loops and conditional select instructions with the original SPARC V8 solution. Using predicated blocks

instead of conditional moves or selects also decreases the number of branches, but causes a performance degradation. Unfortunately, the single-path variant of the *binary greatest common divisor* algorithm is not competitive to the traditional variant, which has a smaller code size on most targets and shows a much better worst-case performance.

Dijkstra Algorithm

The traditional implementation of the *Dijkstra algorithm*, calculating the shortest path between two vertices of a graph, has a code size of 1860 bytes and a worst-case cycle count of 1388 cycles in case of a SPARC V8 instruction set. The optimized single-path variant has a better code size on all targets, but shows much less worst-case performance: In case of the SPARC V8 instruction set, the worst-case cycle count is 2743 cycles, which is about twice the cycle count of the first implementation. Moreover, the resulting assembler code does not have constant execution time, which is only achieved if the target supports conditional blocks. However, the performance decreases for every eliminated branch, resulting in a worst-case execution time of 3325 cycles for targets with predicated blocks based on condition codes and 5071 cycles for targets with predicate registers. Thus, the single-path implementation is only a feasible solution if there is not much memory available and constant execution time is more important than the actual performance.

Interpolation Table

The situation is slightly better for the single-path variant of the *interpolation table*: If the target supports conditional move or select instructions combined with hardware loops, the resulting worst-case cycle count is only about 20 cycles (16 %) higher than the traditional implementation on the SPARC V8 target. The code size of the single-path variant is about 18 % greater in case of no instruction set extensions. When introducing conditional select instructions, the difference gets smaller and code size is only 5 % more than the original solution.

Shell Sort

When analyzing the results of the *shell sort* algorithm evaluation, it can be seen that this is the only example which does not show constant execution times in the single-path transformation. Only when introducing predicated blocks based on predicate registers, which support branch elimination of nested if-then-else blocks, a constant execution time can be achieved. Nevertheless, the resulting worst-case performance of all solutions is about three times the cycle count of the original implementation, meaning that this single-path variant is not very useful.

Software Division

The *software division* algorithm has been implemented to provide a division operation for 16-bit unsigned integers if it is not available in hardware. The first version just subtracts the divisor from the dividend as long as possible and increments a counter. It shows a good average case performance of only a few cycles, but takes over 4 million cycles (!) in the worst case. The second variant uses a combination of shifting and subtraction. Although the code size has increased

for about 55 %, the resulting worst-case cycle count is only 398 cycles for the original SPARC V8 target and 314 cycles if conditional select instructions are provided. The last version is a single-path implementation of the second version. The worst-case cycle count of the SPARC V8 target has increased, but when introducing conditional select instructions and hardware loops, it shows a constant cycle count of 302 cycles. The price of the performance improvement is an increase in code size of more than 100 % which is caused by loop unrolling.

Threshold

Another example which demonstrates the positive influence of the proposed instruction set extensions is the *threshold* algorithm, which can be useful in signal processing applications. It simply checks every value of a given array whether it is below or above a specified threshold. If that is the case, the corresponding element is set to a defined minimum or maximum value. The first version shows the same performance on all evaluated targets; the differences in the code sizes are caused by branch eliminations in the main function. The second solution is the single-path transformation of the algorithm and does not show good performance on the original SPARC V8 instruction set: The code size has slightly increased and the worst-case cycle count is about 50 % higher than the first version. However, if the target supports conditional select and hardware loop instructions, the code size may be even decreased by a few bytes, while the constant execution time is lower than the original worst-case cycle count. Unfortunately, this is not true for any other combination of instruction set extensions.

Instructions Suitable for Time Predictable Processors

Which of the evaluated targets can be seen to be the most suitable solution for real-time systems? It should be a target which shows

- small code size,
- few branches,
- a small deviation (high predictability) and
- a good worst-case performance, i.e., a low worst-case cycle count.

Although there is no target which is outstanding in all of the listed categories, two targets may be seen as a remarkable improvement to the original SPARC V8 instruction set: *v8-movcc* and *v8-selcc*.

Figure 6.5 shows the relative code size of these two targets in comparison with SPARC V8: As can be seen easily, the solution with conditional select instructions is slightly better, meaning that the resulting assembler code is smaller for this target. Moreover, both solutions have a smaller code size than the original solution except for the *bubble sort* algorithm. Nevertheless, the code bloating occurs in the testing function and not in the sorting algorithm itself such that this issue can be neglected.

Concerning *timing predictability*, both targets behave nearly equally, meaning that the deviation of them is almost that of the original SPARC V8 target as can be seen in Figure 6.6. However, for most of the single-path solutions, *v8-movcc* and *v8-selcc* have no deviation at all whereas constant execution time cannot be achieved on the original SPARC V8 instruction set.

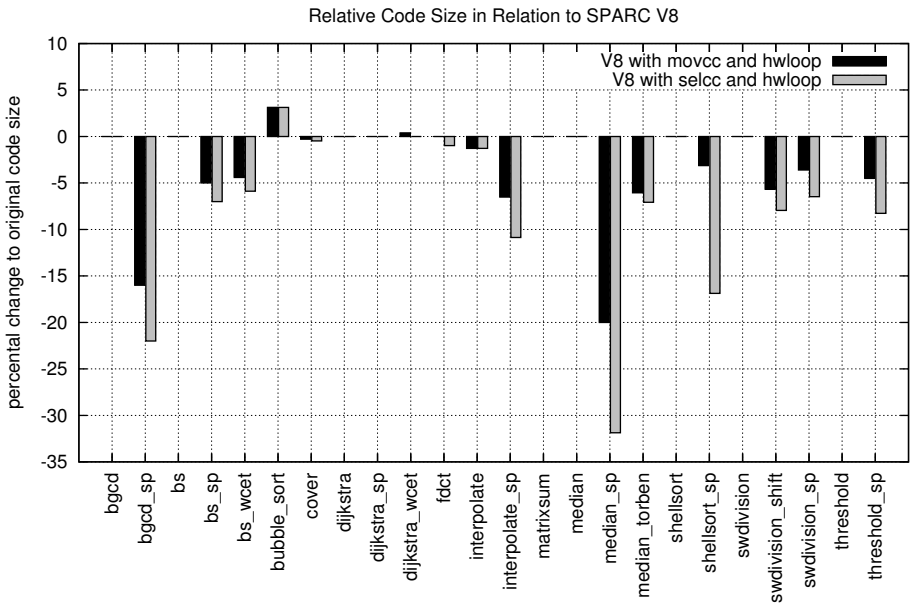

Figure 6.5: Relative code size of *v8-movcc* and *v8-selcc* with original SPARC V8, which corresponds to 100 %.

An important issue for the static analysis of the control flow graph is the number of conditional branches, which are shown in Figure 6.7: In 15 out of 24 algorithms, at least one conditional branch could be replaced either by a conditional move or select instruction or by a hardware loop. The number of branches is equal for the remaining algorithms. Thus, introducing the new instructions either does not change the number of branches at all or makes control flow analysis easier when branches can be removed. In no cases, additional branches have been introduced.

One of the most important factors when designing real-time systems is the worst-case execution time: If a processor shows an excellent average performance, but has a very high worst-case cycle count, another target with lower maximum timings might be preferred. Figure 6.8 shows the relative worst-case cycle count of *v8-movcc* and *v8-selcc* in comparison with the *v8* target: In case of conditional select instructions, the cycle count can be reduced by about 40 % for three algorithms. *v8-movcc* shows similar improvements like *v8-selcc*, but has a much higher worst-case cycle count for *shell sort*. The reason for this lies in the difference of conditional move and conditional select: The latter can be translated easily by LLVM whereas conditional move instructions might involve additional overhead resulting in greater code size and higher execution times. Nevertheless, both targets can be seen as a suitable solution for real-time systems which decrease the code size, the number of branches and the worst-case cycle count in nearly all of the evaluated single-path algorithms.

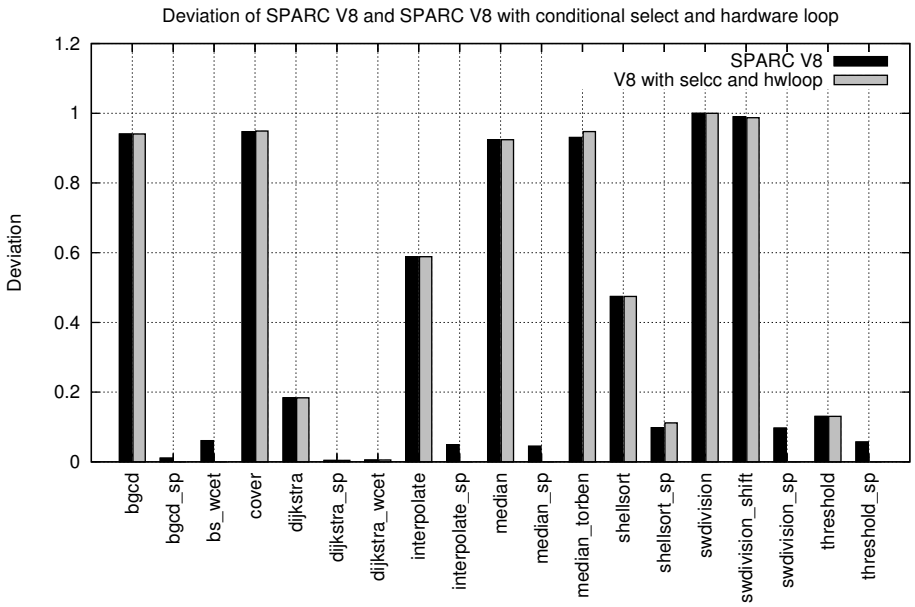

Figure 6.6: Deviation as defined in Equation 6.1 is an indicator of *timing predictability*. A value near 1 denotes bad predictability or high deviation whereas a value near 0 corresponds to nearly constant execution time. The chart shows the deviation of selected algorithms simulated on the SPARC V8 and *v8-selcc* targets.

Summary

Summarizing the results of the evaluated benchmarks shows:

(1) All single-path implementations of the given algorithms show worse performance in comparison with the original version in case none of the instruction set extensions are used.
(2) Introducing several instruction set extensions may increase the code size and the worst-case cycle count for non-single-path algorithms or does not have any effect.
(3) Introducing conditional select and hardware loop instructions increases the worst-case performance of all single-path implementations except for the *Dijkstra algorithm*[5] in comparison with the original SPARC V8 instruction set.
(4) A constant execution time of all single-path algorithms may be achieved by just introducing conditional move or select instructions. This is not true for the shell sort algorithm.
(5) Predicated blocks are not commonly used if conditional move or select instructions are enabled. They cause an increase in code size and show a greater worst-case cycle count than all other solutions.

[5]For the *Dijkstra algorithm*, the compiler has not been able to introduce any of the proposed instruction set extensions such that the assembler code is the same as for the original solution.

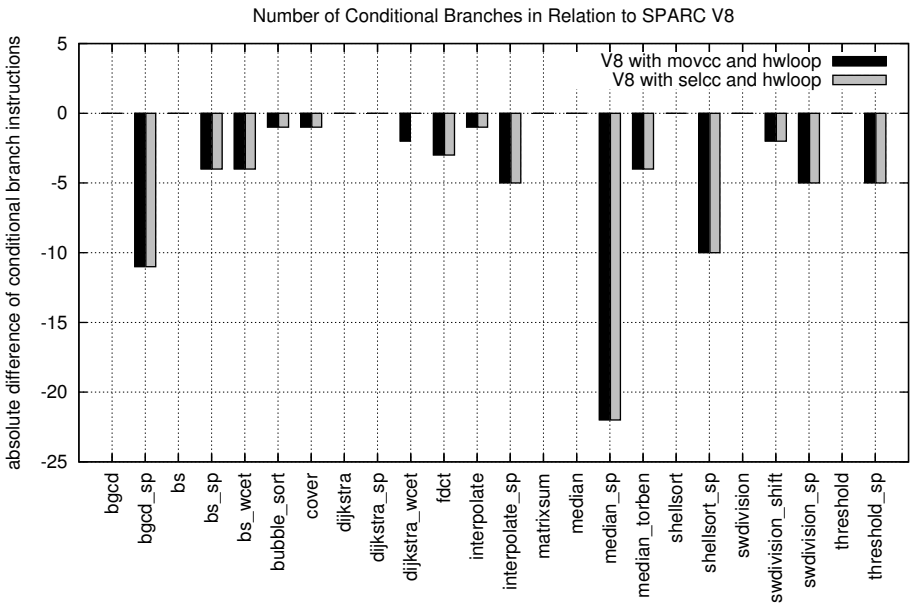

Figure 6.7: Absolute difference in the number of branches of *v8-movcc* and *v8-selcc* in comparison with the number of branches on the original SPARC V8.

(6) Using predicate registers instead of the current processor status register as condition for predicated blocks does not really increase the number of eliminated branches for most cases.
(7) Introducing the hardware loop instruction as the single instruction set extension is nearly useless, because the current compiler implementation is not able to identify simple loops ranging over multiple basic blocks.

Finding the most appropriate combination of instruction set extensions which increase the timing predictability of a processor is based on the following considerations: The hardware costs should be as low as possible, but the resulting worst-case performance should be much better than the performance of the original instruction set. Moreover, the number of eliminated branches and the code size should also be taken into account. Of course, there is no combination of instruction set extensions outperforming all others in every aspect. Nevertheless, two promising combination candidates have been identified to be part of future time-predictable instruction sets: Conditional move instructions with hardware loop support or conditional select with hardware loop support. Both targets show a better worst-case performance for single-path solutions and have a smaller code size than the original SPARC V8 instruction set. Moreover, the resulting assembler code of both targets has fewer conditional branches and is therefore easier to analyze. The decision whether conditional move or select instruction should be implemented mainly depends on the hardware costs. If code size and performance is more important than the number

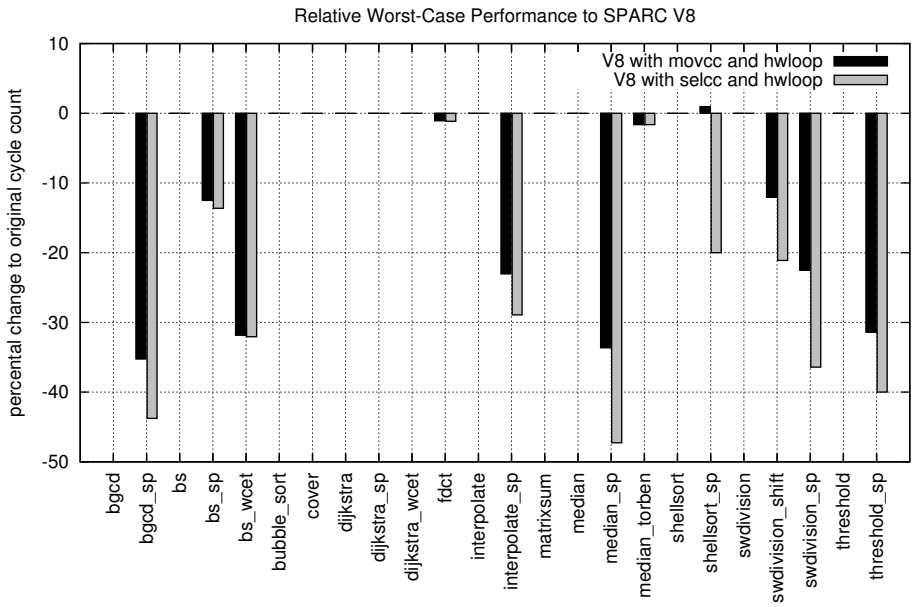

Figure 6.8: Relative worst-case cycle count of *v8-movcc* and *v8-selcc* with original SPARC V8, which corresponds to 100 %. In case no worst-case scenario has been identified, the measured maximum cycle count has been taken.

of used logical elements, the conditional select instruction should be preferred. The hardware loop is a nice-to-have feature, which does not really influence the resulting code size, but may increase the performance by 10 to 20 % while not needing much additional hardware.

CHAPTER 7

Conclusion

7.1 Final Review on the Presented Instruction Set Extensions

It could be seen in the last chapter that conditional move and select instructions are very attractive extensions to existing instruction sets: The additional hardware costs are quite moderate, adaptations of existing compilers are relatively simple and the resulting assembler code shows fewer conditional branches and better performance in many cases. However, the following facts have to be considered for the current evaluation:

- The SPARC V8 instruction set offers several possibilities to be extended, because there is an amount of unused opcodes available. Other targets might not provide such a flexibility for instruction set extensions.
- LLVM natively supports a conditional select instruction. Thus, the code generating process for the LLVM IR language favors the use of the select instruction.
- The performance of the instruction set extensions depends on the quality of generated assembler code. Consequently, improving the code generating passes presented in Chapter 5 might also make other instruction set extensions like predicated blocks more feasible.

Thus, the specific results of the evaluation are only valid for the presented approach using LLVM and the SPARC V8 instruction set. However, in many previous proposals for additional instructions of existing processors (e.g., [TG06, GTS07]), the presented code examples and benchmark algorithms have been implemented manually, resulting in a better performance than the original solution. In the current thesis, all evaluations were based on automatically generated assembler codes. Thus, the presented approach is a proof of concept that even a compiler which has not been designed to generate predictable assembler code is able to correctly and efficiently translate single-path algorithms if the underlying instruction set provides corresponding instructions.

7.2 Suggestions for Further Research

The current evaluation is based on the measured results of a SPARC V8 instruction set simulator. More realistic results can be obtained if the proposed instruction set extensions will really be implemented in hardware. Moreover, the presented code generating algorithms could be improved such that hardware loops may cover multiple basic blocks, nested loops can be translated and branch elimination is done more efficiently. All additional compiler passes use assembler code input and try to identify branches and loops. If additional information from high-level programming languages is used, code generation will become easier and perhaps more efficient.

Another interesting investigation could include the performance evaluation on targets which provide predicated and conditional move instructions. Small adaptations to the compiler could generate assembler output using conditional move instructions without any other predications and compare the execution times to assembler code without any restrictions. Possible processors could include ARM or the Intel Itanium.

A complete different approach is implemented by VLIW processors: Their ISA is completely different to the SPARC V8 and would need other types of instructions to provide predictability. There are several investigations on creating predictable VLIW architectures, e.g., [SSP+11]. VLIW offers the possibility to execute the if- and else-branch concurrently, which are ideal preconditions for a single-path transformation.

7.3 Summary

The current thesis has given an overview of existing processors used in real-time systems. Moreover, several approaches for time predictable architectures have been presented. Unfortunately, most of them do not focus on the underlying instruction set, but either on the complete system design or on the specific implementation in hardware. In this thesis, several instructions have been identified which should be part of future processors if they want to provide *timing predictability*: It is not only sufficient to offer constant execution times of hardware instructions, but also to provide instructions which allow the compiler to produce *time predictable code*.

The SPARC V8 instruction set, which has been designed to fit for a general purpose processor, has been extended such that predictable assembler code with constant execution times can be generated. The LLVM compiler framework has been adapted to support the newly introduced instructions. It has been shown that a number of conditional branches can be replaced easily if the underlying instruction set supports conditional execution of instructions (e.g., predicated blocks) and hardware loops.

Finally, the impacts of the instruction set extensions on the generated assembler code and the performance have been evaluated by simulating more than 20 benchmark algorithms on targets providing different instruction sets. It has been shown that a conditional move instruction is sufficient to generate single-path code of an algorithm and that this solution shows better performance, i.e., lower execution time, than introducing a predicated instruction set. The most promising combinations of the presented instruction set extensions have been identified and suggested to be part of future time predictable instruction sets. These include conditional move or select instructions together with support of hardware loops: For most of the evaluated benchmark

algorithms the code size has decreased, the number of basic blocks and conditional branches has been reduced and the worst-case timing behavior has improved.

Appendices

APPENDIX A

Performance Evaluation of Selected Algorithms

All benchmark algorithms of the current appendix have been first translated into the LLVM IR language by `llvm-gcc`. If not stated differently, all compiler optimizations have been enabled (`-O3`). Afterwards, the LLVM compiler `llc` has been used to generate SPARC V8 assembler code with certain instruction set extensions. The implemented instruction set of each target can be found in Section 6.2. Due to a bug, the elimination of the index variable in hardware loops had to be generally disabled.

A.1 Bubble Sort

The original implementations of the bubble sort algorithm may be found in [Pus07, p. 7-19]. The evaluated scenarios include a sorted array, equivalent to the best case, a pseudo-randomly initialized array and an array sorted in reverse order, which is the worst case for all variants. The array consisted of 100 integer elements. The version numbering corresponds to the numbering of the paper.

	Version 1	Version 2	Version 4	Version 5	Version 6
v8	368	388	376	396	440
v8-predblockicc	368	392	376	396	444
v8-hwloop	368	388	376	396	452
v8-movcc	368	384	376	396	440
v8-selcc	368	380	376	396	428

Table A.1: Code size of several implementations of bubble sort on different targets. The given numbers represent the code size in bytes.

	Version 1		Version 2		Version 4		Version 5		Version 6	
	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET
v8	70 194	99 894	105 042	114 942	1 398	104 946	1 398	105 339	23 267	125 763
v8-predblockicc	99 894	99 894	119 892	119 892	2 091	104 946	2 091	105 339	23 465	139 987
v8-hwloop	70 194	99 894	105 042	114 942	1 398	104 946	1 398	105 339	14 251	125 763
v8-movcc	70 194	99 894	87 420	87 420	1 398	104 946	1 398	105 339	92 764	92 764
v8-selcc	70 194	99 894	82 371	82 371	1 398	104 946	1 398	105 339	82 666	82 666

Table A.2: Performance evaluation of several implementations of bubble sort on different targets. All given numbers represent the minimum (BCET) or maximum (WCET) cycle count for the specified version. The test input vectors were arrays of 100 integers.

A.2 Find First

The original implementations of the find first algorithm may be found in [Pus07, p. 21-28]. The evaluated scenarios were taken from [Pus07, p. 62f] and cover single, multiple and no occurences of the key in an array of 10 integers. The version numbering corresponds to the numbering of the paper.

	Version 1	**Version 2**	**Version 3a**	**Version 3b**	**Version 5/6a**	**Version 5/6b**
v8	152	180	768	216	304	168
v8-predblockicc	152	184	892	228	312	172
v8-hwloop	152	180	768	216	304	168
v8-movcc	152	164	604	188	272	156
v8-selcc	152	160	484	172	228	156

Table A.3: Code size of several implementations of find first on different targets. The given numbers represent the code size in bytes. Versions 3 and 5/6 were generated with different `llvm-gcc` settings: (a) is the assembler output if all optimizations are turned on, (b) is the assembler output if loop unrolling is disabled.

	Version 1		Version 2		Version 3a		Version 3b		Version 5/6a		Version 5/6b	
	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET
v8	12	107	15	172	152	159	250	259	47	55	161	171
v8-predblockicc	12	107	17	192	200	200	293	293	57	57	181	181
v8-hwloop	12	107	15	172	152	159	250	259	47	55	161	171
v8-movcc	12	107	11	132	129	129	147	147	48	48	141	141
v8-selcc	12	107	11	132	101	101	126	126	40	40	141	141

Table A.4: Performance evaluation of several implementations of find first on different targets. All given numbers represent the minimum (BCET) or maximum (WCET) cycle count for the specified version. Versions 3 and 5/6 were generated with different `llvm-gcc` settings: (a) is the assembler output if all optimizations are turned on, (b) is the assembler output if loop unrolling is disabled.

A.3 Binary Search

The original implementations of the binary search algorithm may be found in [Pus07, p. 30-52]. The evaluated scenarios use a logarithmically distributed integer array of 16 elements. The keys to be found are saved in the array or are greater than the largest element, which is the worst-case scenario. The version numbering corresponds to the numbering of the paper.

	Version 1/2	**Version 3**	**Version 5**	**Version 7**	**Version 8**	**Version 10**
v8	208	288	220	216	244	224
v8-predblockicc	208	288	220	216	244	224
v8-hwloop	208	288	220	216	244	224
v8-movcc	196	256	212	196	228	212
v8-selcc	196	240	208	196	224	200

Table A.5: Code size of several implementations of binary search on different targets. The given numbers represent the code size in bytes. For all versions, loop unrolling has been disabled.

	Version 1/2		Version 3		Version 5		Version 7		Version 8		Version 10	
	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET	BCET	WCET
v8	10	101	85	161	79	85	74	75	88	97	78	79
v8-predblockicc	10	106	93	181	90	90	79	79	93	97	83	83
v8-hwloop	10	101	85	161	79	85	74	75	88	97	78	79
v8-movcc	10	91	77	145	68	68	62	62	80	84	62	62
v8-selcc	10	91	65	129	67	67	174	174	79	83	57	57

Table A.6: Performance evaluation of several implementations of binary search on different targets. All given numbers represent the minimum (BCET) or maximum (WCET) cycle count for the specified version. Loop unrolling has been generally disabled. Note that version 7 has a very low performance on the *v8-selcc* target, which is even worse than the performance on the *v8-predblockicc* target. This is due to a bug in the hardware loop code generating pass, which inserts a hardware loop where it should not be done. When disabling hardware loops, the resulting constant execution time is 61 cycles.

A.4 Increment Multi-byte Counter

The original implementations of the binary search algorithm may be found in [Pus07, p. 54-58]. The evaluated scenarios use an 8-byte counter, initially set to the largest value. It will be incremented for 65 535 times. The version numbering corresponds to the numbering of the paper.

	Version 1	**Version 2**	**Version 4**
v8	140	164	144
v8-predblockicc	140	164	144
v8-hwloop	140	164	144
v8-movcc	140	164	148
v8-selcc	140	152	140

Table A.7: Code size of several implementations of incrementing a multi-byte counter on different targets. The given numbers represent the code size in bytes. For all versions, loop unrolling has been disabled.

	Version 1		**Version 2**		**Version 4**	
	BCET	WCET	BCET	WCET	BCET	WCET
v8	16	142	182	191	152	154
v8-predblockicc	16	142	202	202	161	161
v8-hwloop	16	142	182	191	152	154
v8-movcc	16	142	156	156	135	135
v8-selcc	16	142	145	145	115	115

Table A.8: Performance evaluation of several implementations of incrementing a multi-byte counter on different targets. All given numbers represent the minimum (BCET) or maximum (WCET) cycle count for the specified version. Loop unrolling has been generally disabled.

APPENDIX B

Benchmark Results

All benchmark algorithms of the current appendix have been first translated into the LLVM IR language by `llvm-gcc`. All compiler optimizations have been enabled (`-O3`). Afterwards, the LLVM compiler `llc` has been used to generate SPARC V8 assembler code with certain instruction set extensions. To evaluate which combinations of additional instructions are most feasible, 13 different targets have been defined:

- **v8** – The original SPARC V8 instruction set without any extensions.
- **v8-m** – The original SPARC V8 instruction set with conditional move instructions.
- **v8-s** – The original SPARC V8 instruction set with conditional select instructions.
- **v8-i** – The original SPARC V8 instruction set with predicated blocks based on integer condition codes.
- **v8-r** – The original SPARC V8 instruction set with predicated blocks based on predicate registers.
- **v8-l** – The original SPARC V8 instruction set with additional support for hardware loops.
- **v8-ml** – The original SPARC V8 instruction set with conditional move instructions and support for hardware loops.
- **v8-sl** – The original SPARC V8 instruction set with conditional select instructions and support for hardware loops.
- **v8-il** – The original SPARC V8 instruction set with predicated blocks based on integer condition codes and support for hardware loops.
- **v8-rl** – The original SPARC V8 instruction set with predicated blocks based on predicate registers and support for hardware loops.
- **v8-mil** – The original SPARC V8 instruction set with conditional move instructions, predicated blocks based on integer condition codes and support for hardware loops.
- **v8-srl** – The original SPARC V8 instruction set with conditional select instructions, predicated blocks based on predicate registers and support for hardware loops.
- **v8-sil** – The original SPARC V8 instruction set with conditional select instructions, predicated blocks based on integer condition codes and support for hardware loops.

For each algorithm, the following key figures have been recorded:

- *Code Size* is the code size of the complete program, including test and main functions, in bytes.
- *Conditional Branches* denotes the number of conditional branches within the complete program. It is an indication of the complexity of the control flow graph. A lower number means that the complexity of the control-flow graph decreases and the worst-case path is easier to identify.
- *Unconditional Branches* denote the number of jump instructions within the complete program. They usually do not have an influence on the complexity of the control flow graph.
- *Number of NOPs* denotes the number of delay instructions which have to be inserted after a conditional or unconditional branch, see Section 4.2.2 for details.
- *Number of MBBs* is the amount of machine basic blocks of the whole program and corresponds to the number of vertices of the control flow graph. Like the for conditional branches, a lower number is an indicator that the worst-case path of the given assembler code is easier to identify.
- *Min/Max Cycles* is the measured minimum/maximum cycle count and does not need to be the actual worst-case execution time. For algorithms with a single vector of input values, only the measured cycle count has been recorded.
- *Deviation* is calculated by the formula specified in Equation 6.1 and is only stated if the minimum and maximum cycle count of the current algorithm are available.
- *Number of movCCs* denotes the number of conditional move instructions which have been used within the whole program.
- *Number of selCCs* denotes the number of conditional select instructions which have been used within the whole program.
- *Number of predbegins* is equivalent to the number of basic blocks which have been predicated.
- *Nesting level of predicated blocks* is only available for predicated instructions based on predicate registers. It is equivalent to the highest predicate register which has been used within the whole program. This relates to the nesting level of if-then-else structures which could be eliminated.
- *Number of HWLoops* denotes the number of used hardware loops within the whole program.

B.1 Binary Greatest Common Divisor

This implementation of the greatest common divisor algorithm by Euclid does neither use modulo nor division operations. All calculations are based on addition, subtraction and shift operations, which are part of most instruction sets, see the code listing below. As can be seen in the following table, only two basic blocks can be eliminated when predicated blocks are supported. In all other cases, none of the presented instruction set extensions is used, resulting that the performance of all targets is quite similar.

```
#define EVEN(x) ((x&0x1)==0)
```

```
static int gcd_binary(int a, int b)
{
     unsigned int u = (a < 0) ? (-a) : a;
     unsigned int v = (b < 0) ? (-b) : b;
     int shift= 0;

     if (u == 0 || v == 0) { return (u|v); }

     while(EVEN(u)) {
        u >>= 1;
        if(EVEN(v)) {
          v >>= 1;
          shift++;
        }
     }
     while(v > 0) {
       while(EVEN(v)) v >>= 1;
       if(u < v) v -= u;
       else {
            int diff = u-v;
            u = v;
            v = diff;
       }
     }
     return u << shift;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	304	304	304	304	312	304	304	304	304	312	304	312	304
Conditional Branches	11	11	11	10	10	11	11	11	10	10	10	10	10
Unconditional Branches	5	5	5	4	4	5	5	5	4	4	4	4	4
Number of NOPs	9	9	9	8	8	9	9	9	8	8	8	8	8
Number of MBBs	23	23	23	21	21	23	23	23	21	21	21	21	21
Max Cycles	554	554	554	615	665	554	554	554	615	665	615	665	615
Min Cycles	33	33	33	35	37	33	33	33	35	37	35	37	35
Deviation	0.9404	0.9404	0.9404	0.9431	0.9444	0.9404	0.9404	0.9404	0.9431	0.9444	0.9431	0.9444	0.9431
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	2	2	-	-	-	2	2	2	2	2
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	0	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.2 Binary Greatest Common Divisor – Single-Path

This is the single-path implementation of the binary version of the Euclid algorithm. As can be seen in the following table, nearly all branches can be removed if the target supports conditional move or select instructions or predicated blocks. Note that the number of basic blocks increases if hardware loops are introduced. However, their usage reduces the amount of conditional branches, such that the complexity of the CFG decreases.

```
#define EVEN(x) ((x&0x1)==0)

static int gcd_binary(int a, int b)
{
     unsigned int u = (a < 0) ? (-a) : a;
     unsigned int v = (b < 0) ? (-b) : b;
     int shift= 0;

     if(u == 0 || v == 0) { u=v=(u|v); }

     int i;
     for(i = 0; i < 31; i++) {
       int even_u = EVEN(u);
       int even_v = EVEN(v);
       if(even_u) u >>= 1;
       if(even_v) v >>= 1;
       if(even_u & even_v) shift++;
     }
     for(i = 0; i < 60; ++i ) {
       int diff = u-v;
       if( (v & 0x1) & (diff < 0)) v = -diff;
       if( (v & 0x1) & (diff >= 0)) u = v;
       if( (v & 0x1) & (diff >= 0)) v = diff;
       v >>= 1;
     }
     return u << shift;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	400	328	308	412	484	404	336	312	420	492	336	312	312
Conditional Branches	14	5	5	5	5	13	3	3	3	3	3	3	3
Unconditional Branches	2	1	1	1	1	2	1	1	1	1	1	1	1
Number of NOPs	8	2	3	2	2	8	2	2	2	2	2	2	2
Number of MBBs	30	11	11	11	11	31	13	13	13	13	13	13	13
Max Cycles	2114	1634	1514	2354	2954	2025	1369	1189	2089	2689	1369	1189	1189
Min Cycles	2091	1634	1514	2354	2954	2002	1369	1189	2089	2689	1369	1189	1189
Deviation	0.0109	0	0	0	0	0.0114	0	0	0	0	0	0	0
Number of movCCs	-	9	-	-	-	-	9	-	-	-	9	-	-
Number of selCCs	-	-	9	-	-	-	-	9	-	-	-	9	9
Number of predbegins	-	-	-	10	10	-	-	-	10	10	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	1	2	2	2	2	2	2	2

B.3 Binary Search

The first implementation of the binary search algorithm is similar to the traditional implementation presented in Section 6.2.3. As the input vectors for the evaluation differ in size, the deviation of maximum and minimum execution time has not been calculated. It can be seen in the following table that none of the presented instruction set extensions could be used.

```
static int bs(int*keys, int key, int size) {
  int mid;
  int low  = 0;
  int high = size - 1;
  while (low <= high) {
    mid = (int) (((unsigned int)low + (unsigned int)high) >> 1);
    int midVal = keys[mid];

    if (midVal < key)           low = mid + 1;
    else if (midVal > key)      high = mid - 1;
    else                        return mid; // key found
  }
  return -1;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	248	248	248	248	248	248	248	248	248	248	248	248	248
Conditional Branches	8	8	8	8	8	8	8	8	8	8	8	8	8
Unconditional Branches	4	4	4	4	4	4	4	4	4	4	4	4	4
Number of NOPs	5	5	5	5	5	5	5	5	5	5	5	5	5
Number of MBBs	16	16	16	16	16	16	16	16	16	16	16	16	16
Max Cycles	96	96	96	96	96	96	96	96	96	96	96	96	96
Min Cycles	23	23	23	23	23	23	23	23	23	23	23	23	23
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	0	0	-	-	-	0	0	0	0	0
Nesting level of predicated blocks	-	-	-	-	-	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.4 Binary Search – Single-Path for Fixed Size

This first single-path implementation of the binary search algorithm is equivalent to version 5 in [Pus07]. The execution time for arrays of equal sizes is constant, but as can be seen in the following table, small arrays have a lower cycle count. Thus, the current implementation may be only seen as a single-path solution for arrays with fixed sizes.

```
static int bs(int*keys, int key, int size)
{
  int inc;
  int left = 0, right = size-1;
  int idx = (right + left) >> 1;
  for(inc = size; inc > 0; inc >>= 1) {
    right = (key < keys[idx] ? idx-1 : right);
    left  = (key > keys[idx] ? idx+1 : left);
    idx   = (right + left) >> 1;
  }
  return (keys[idx] == key) ? idx : -1;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	400	380	372	404	444	400	380	372	404	444	380	380	372
Conditional Branches	14	10	10	10	9	14	10	10	10	9	10	9	10
Unconditional Branches	3	3	3	3	2	3	3	3	3	2	3	2	3
Number of NOPs	9	6	6	6	5	9	6	6	6	5	6	5	6
Number of MBBs	25	17	17	17	14	25	17	17	17	14	17	14	17
Max Cycles	88	77	76	93	115	88	77	76	93	115	77	76	76
Min Cycles	26	25	24	29	35	26	25	24	29	35	25	24	24
Number of movCCs	-	4	-	-	-	-	4	-	-	-	4	-	-
Number of selCCs	-	-	4	-	-	-	-	4	-	-	-	4	4
Number of predbegins	-	-	-	4	6	-	-	-	4	6	0	2	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	0	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.5 Binary Search – Single-Path for Variable Size

This is an improved version of the first single-path implementation of binary search. As can be seen in the following table, the cycle count has dramatically increased, but is equal for arrays of all sizes. The reason for this is the fact, that the input vectors of the evaluation only consists of 16 elements, but the loop bounds have been chosen to support array sizes of 32 elements. Note that loop unrolling has been enabled for the first single-path solution, causing that the resulting code size is much greater than for the current implementation.

```
static int bs(int*keys, int key, int size)
{
  int i;
  int left = 0, right = size-1;
  int idx = (right + left) >> 1;

  for(i = 0; i < sizeof(int)*8; i++) {
    right = (key < keys[idx] ? idx-1 : right);
    left  = (key > keys[idx] ? idx+1 : left);
    idx   = (right + left) >> 1;
  }
  return (keys[idx] == key) ? idx : -1;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	272	256	252	272	296	272	260	256	276	300	260	256	256
Conditional Branches	9	6	6	6	6	9	5	5	5	5	5	5	5
Unconditional Branches	3	3	3	3	3	3	3	3	3	3	3	3	3
Number of NOPs	7	4	4	4	4	7	4	4	4	4	4	4	4
Number of MBBs	18	12	12	12	12	18	13	13	13	13	13	13	13
Max Cycles	493	428	427	525	655	493	336	335	433	563	336	335	335
Min Cycles	463	428	427	525	655	463	336	335	433	563	336	335	335
Deviation	0.0609	0	0	0	0	0.0609	0	0	0	0	0	0	0
Number of movCCs	-	3	-	-	-	-	3	-	-	-	3	-	-
Number of selCCs	-	-	3	-	-	-	-	3	-	-	-	3	3
Number of predbegins	-	-	-	3	3	-	-	-	3	3	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	-	-
Number of HWLoops	-	-	-	-	-	0	1	1	1	1	1	1	1

B.6 Bubble Sort – Worst Case Scenario

This is a performance test for the worst-case scenario of the bubble sort algorithm. As can be seen in the following table, introducing hardware loops does not change the performance at all, which is due to the fact, that the loop occurs in the initialization function and is therefore not considered in the evaluation. Neither conditional move nor select instructions could be used, but two branches could be eliminated when predicated blocks were enabled. Nevertheless, the resulting cycle count is worse than the original solution for targets using predicate registers.

```
static void BubbleSort(int Array[])
{
   int Sorted = FALSE;
   int Temp, LastIndex, Index, i;

   for (i = 1; i <= NUMELEMS-1; i++)
   {
      Sorted = TRUE;
      for (Index = 1; Index <= NUMELEMS-1; Index ++) {
         if (Index > NUMELEMS-i)
            break;
         if (Array[Index] > Array[Index + 1])
         {
            Temp = Array[Index];
            Array[Index] = Array[Index+1];
            Array[Index+1] = Temp;
            Sorted = FALSE;
         }
      }

      if (Sorted)
         break;
   }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	256	256	256	256	264	264	264	264	264	272	264	272	264
Conditional Branches	9	9	9	8	8	8	8	8	7	7	7	7	7
Unconditional Branches	2	2	2	1	1	2	2	2	1	1	1	1	1
Number of NOPs	6	6	6	5	5	6	6	6	5	5	5	5	5
Number of MBBs	17	17	17	14	14	18	18	18	15	15	15	15	15
Cycles	125138	125138	125138	125138	135038	125138	125138	125138	125138	135038	125138	135038	125138
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	2	2	-	-	-	2	2	2	2	2
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	0	-
Number of HWLoops	-	-	-	-	-	1	1	1	1	1	1	1	1

B.7 Switch Case Test

The current test implements three functions dealing with a loop and a tremendous amount of switch-case instructions. As the loop bounds are different for each function, only the maximum and minimum cycle count have been recorded. The following table shows that this algorithm does not benefit at all from the presented instruction set extensions. The introduced conditional move or select instruction is only part of the evaluation function.

```
static int swi120(int c)
{
  int i;
  for (i=0; i<120; i++) {
    switch (i) {
      case 0: c++; break;
      ...
      case 119: c++; break;
      default: c--; break;
    }
  }
  return c;
}

static int swi50(int c)
{
  int i;
  for (i=0; i<50; i++) {
    switch (i) {
      case 0: c++; break;
      ...
      case 59: c++; break;
      default: c--; break;
    }
  }
  return c;
}

static int swi10(int c)
{
  int i;
  for (i=0; i<10; i++) {
    switch (i) {
      case 0: c++; break;
      ...
      case 9: c++; break;
      default: c--; break;
    }
  }
  return c;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	4164	4152	4144	4164	4164	4164	4152	4144	4164	4164	4152	4144	4144
Conditional Branches	279	278	278	279	279	279	278	278	279	279	278	278	278
Unconditional Branches	88	88	88	88	88	88	88	88	88	88	88	88	88
Number of NOPs	279	278	278	279	279	279	278	278	279	279	278	278	278
Number of MBBs	375	373	373	375	375	375	373	373	375	375	373	373	373
Max Cycles	3390	3390	3390	3390	3390	3390	3390	3390	3390	3390	3390	3390	3390
Min Cycles	179	179	173	179	179	179	179	173	179	179	179	173	173
Number of movCCs	-	1	-	-	-	-	1	-	-	-	1	-	-
Number of selCCs	-	-	1	-	-	-	-	1	-	-	-	1	1
Number of predbegins	-	-	-	0	0	-	-	-	0	0	0	0	0
Nesting level of predicated blocks	-	-	-	-	-	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	-	0	0	0

B.8 Dijkstra Algorithm

The Dijkstra algorithm identifies the shortest path within a given graph. It is one of the few examples, where predicated instructions based on predicated registers is able to remove nested if-then-else structures such that hardware loops are successfully identified. Nevertheless, the resulting solution is neither single-path nor does it show better performance than the other implementations.

```
static void dijkstra( int cost[][NRNODES], int *preced, int *distance)
{
  int selected[NRNODES]={0};
  int current=0,k=0,i,dc,smalldist,newdist;

  for(i=0;i<NRNODES;i++)
    distance[i]=INFINITE;

  selected[current]=1;
  distance[0]=0;
  current=0;

  while(!allselected(selected))
    {
      smalldist=INFINITE;
      dc=distance[current];
      for(i=0;i<NRNODES;i++)
        {
          if(selected[i]==0)
            {
              newdist=dc+cost[current][i];
              if(newdist<distance[i])
                {
                  distance[i]=newdist;
                  preced[i]=current;
                }
              if(distance[i]<smalldist)
                {
                  smalldist=distance[i];
                  k=i;
                }
            }
        }
      current=k;
      k=0;
      selected[current]=1;
    }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	1860	1860	1860	1860	1960	1860	1860	1860	1860	1980	1860	1980	1860
Conditional Branches	40	40	40	30	25	40	40	40	30	20	30	20	30
Unconditional Branches	15	15	15	15	15	15	15	15	15	15	15	15	15
Number of NOPs	44	44	44	34	29	44	44	44	34	24	34	24	34
Number of MBBs	91	91	91	71	61	91	91	91	71	66	71	66	71
Max Cycles	1388	1388	1388	1418	2069	1388	1388	1388	1418	2004	1418	2004	1418
Min Cycles	1133	1133	1133	1253	1904	1133	1133	1133	1253	1802	1253	1802	1253
Deviation	0.1837	0.1837	0.1837	0.1164	0.0797	0.1837	0.1837	0.1837	0.1164	0.1008	0.1164	0.1008	0.1164
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	10	20	-	-	-	10	20	10	20	10
Nesting level of predicated blocks	-	-	-	-	1	-	-	-	-	1	-	1	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	5	0	5	0

B.9 Dijkstra Algorithm – Single-Path

Although this implementation of the Dijkstra algorithm has been written such that single-path code should be generated, LLVM has not been able to do so except for targets with predicated blocks. The number of conditional branches could be decreased on all targets as well as the code size, but the resulting performance is very poor.

```
static void dijkstra(int cost[][NRNODES],int *preced, int *distance)
{
    int selected[NRNODES]={0}, dc;
    volatile int current=0,k=0,i,j,smalldist,newdist,temp, preced_temp
        , dist_temp;

    for(i=0;i<NRNODES;i++)
       distance[i]=INFINITE;

    selected[current]=1;
    distance[0]=0;
    current=0;

    for(j=0;j<NRNODES-1;j++)
    {
        smalldist=INFINITE;
        dc=distance[current];
        for(i=0;i<NRNODES;i++)
        {
            newdist= dc+cost[current][i];

            preced_temp = preced[i];
            dist_temp = distance[i];
            temp = (newdist < dist_temp) ? current : preced_temp;
            preced[i]=(selected[i]==0) ? temp : preced_temp;
            temp = (newdist < dist_temp) ? newdist : dist_temp;
            distance[i]=(selected[i]==0) ? temp : dist_temp;

            dist_temp = distance[i];
            temp = (dist_temp < smalldist) ? i : k;
            k=(selected[i]==0) ? temp : k;
            temp = (dist_temp < smalldist) ? dist_temp : smalldist;
            smalldist=(selected[i]==0) ? temp : smalldist;
        }
        current=k;
        selected[current]=1;
   }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	1236	1236	1236	1244	1308	1236	1236	1236	1244	1308	1244	1308	1244
Conditional Branches	24	24	24	16	16	24	24	24	16	16	16	16	16
Unconditional Branches	13	13	13	5	5	13	13	13	5	5	5	5	5
Number of NOPs	26	26	26	20	20	26	26	26	20	20	20	20	20
Number of MBBs	58	58	58	34	34	58	58	58	34	34	34	34	34
Max Cycles	7160	7160	7160	8072	8744	7160	7160	7160	8072	8744	8072	8744	8072
Min Cycles	7130	7130	7130	8071	8743	7130	7130	7130	8071	8743	8071	8743	8071
Deviation	0.0042	0.0042	0.0042	0.0001	0.0001	0.0042	0.0042	0.0042	0.0001	0.0001	0.0001	0.0001	0.0001
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	16	16	-	-	-	16	16	16	16	16
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	0	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.10 Dijkstra Algorithm – Optimized Single-Path

The optimized single-path version of the Dijkstra algorithm still is much slower than the original solution and only shows constant execution time on targets providing predicated instructions. However, the code size and number of conditional branches could again be reduced. Thus, the current implementation is superior to the first single-path solution.

```
static void dijkstra(int cost[][NRNODES],int *preced,int *distance)
{
    int selected[NRNODES]={0}, dc;
    volatile int current=0,k=0,i,j,smalldist,newdist;
    volatile int preced_temp, dist_temp;

    for(i=1;i<NRNODES;i++)
       distance[i]=INFINITE;

    selected[current]=1;
    distance[0]=0;

    for(j=0;j<NRNODES-1;j++) {
        smalldist=INFINITE;
        dc=distance[current];

        for(i=1;i<NRNODES;i++) {

            if(selected[i]==0) {
                newdist= dc+cost[current][i];
                preced_temp = preced[i];
                dist_temp = distance[i];

                if(newdist < dist_temp) {
                    preced[i] = current;
                    distance[i] = newdist;
                    dist_temp = newdist;
                } else {
                    preced[i] = preced_temp;
                    distance[i] = dist_temp;
                    dist_temp = dist_temp;
                }

                if(dist_temp < smalldist) {
                    k = i;
                    smalldist = dist_temp;
                } else {
                    k = k;
                    smalldist = smalldist;
                }
            }
        }
        current=k;
        selected[current]=1;
    }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	1084	1084	1084	1088	1112	1084	1088	1084	1088	1112	1088	1112	1088
Conditional Branches	19	19	19	17	16	19	17	19	17	16	17	16	17
Unconditional Branches	7	7	7	5	5	7	5	7	5	5	5	5	5
Number of NOPs	22	22	22	21	20	22	21	22	21	20	21	20	21
Number of MBBs	42	42	42	36	34	42	36	42	36	34	36	34	36
Max Cycles	2743	2743	2743	3325	5071	2743	2743	2743	3325	5071	3325	5071	3325
Min Cycles	2728	2728	2728	3324	5070	2728	2728	2728	3324	5070	3324	5070	3324
Deviation	0.0055	0.0055	0.0055	0.0003	0.0002	0.0055	0.0055	0.0055	0.0003	0.0002	0.0003	0.0002	0.0003
Number of movCCs	-	0	-	-	-	-	-	-	-	-	-	-	-
Number of selCCs	-	-	0	-	-	0	-	0	-	-	-	0	0
Number of predbegins	-	-	-	4	7	-	4	-	4	7	4	7	4
Nesting level of predicated blocks	-	-	-	-	1	-	-	-	-	1	-	1	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.11 Fourier Discrete Cosine Transformation

This algorithm implements the Fourier discrete cosine transformation of a 8×8 matrix. Although all conditional branches can be removed if hardware loops and conditional move or select instructions are provided, the performance only slightly increases. This is due to the fact that the algorithm involves a lot of arithmetic operations, which are usually part of DSPs. Hence, introducing SIMD instructions could reduce the cycle count more than the proposed instruction set extensions.

```
static void fdct(short int *blk, int lx)
{
  int tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  int tmp10, tmp11, tmp12, tmp13;
  int z1, z2, z3, z4, z5;
  int i;
  short int *block;

  int constant;

  /* Pass 1: process rows. */
  /* Note results are scaled up by sqrt(8) compared to a true DCT; */
  /* furthermore, we scale the results by 2**PASS1_BITS. */

  block=blk;

  for (i=0; i<8; i++)
  {
    tmp0 = block[0] + block[7];
    ...
    block[7] = (tmp4 + z1 + z3) >> (CONST_BITS-PASS1_BITS);
    ...
    block += lx;
  }

  /* Pass 2: process columns. */

  block=blk;

  for (i = 0; i<8; i++)
  {
    tmp0 = block[0] + block[7*lx];
    ...
    block[2*lx] = (z1 + (tmp13 * constant)) >> (CONST_BITS+PASS1_BITS
        +3);
    ...
    /* advance to next column */
    block++;
  }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	820	808	800	820	820	832	820	812	832	832	820	812	812
Conditional Branches	3	2	2	3	3	1	0	0	1	1	0	0	0
Unconditional Branches	0	0	0	0	0	0	0	0	0	0	0	0	0
Number of NOPs	2	1	1	2	2	1	0	0	1	1	0	0	0
Number of MBBs	6	4	4	6	6	8	6	6	8	8	6	6	6
Cycles	2956	2956	2954	2956	2956	2924	2924	2922	2924	2924	2924	2922	2922
Number of movCCs	-	1	-	-	-	-	1	-	-	-	1	-	-
Number of selCCs	-	-	1	-	-	-	-	1	-	-	-	1	1
Number of predbegins	-	-	-	0	0	-	-	-	0	0	0	0	0
Nesting level of predicated blocks	-	-	-	-	-	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	2	2	2	2	2	2	2	2

B.12 Interpolation Table

This algorithm implements the calculation of a mathematical function of which only a few values are known. The output value is calculated by interpolation between the two nearest available values. As can be seen in the following table, the instruction set extensions are only used sparely and do not have much influence on the performance.

```
static int16_t tab_lookup(const tab * map, int16_t x)
{
   int16_t Aux_S16, Aux_S16_a;
   uint16_t Aux_U16, Aux_U16_a, t;

   const int16_t * x_table ;
   const int16_t * z_table ;

   x_table = (const int16_t *) map->x_table;
   z_table = (const int16_t *) map->z_table;

   if (x <= *(x_table)) {
      return z_table[0];
   }

   if (x >= x_table[(uint8_t)(map->Nx - 1)]) {
      return z_table[(uint8_t) (map->Nx - 1)];
   }

   (x_table)++;
   while (x > *((x_table)++)) {
      (z_table)++;
   }
   x_table -= 2 ;
   Aux_S16 = *((z_table)++);
   Aux_S16_a = *(z_table);


   Aux_U16 = (uint16_t) (((uint16_t) x) - ((uint16_t) x_table[0]));
   Aux_U16_a = (uint16_t) (((uint16_t) x_table[1]) - ((uint16_t)
       x_table[0]));

   ...

   if (Aux_S16 <= Aux_S16_a) {
      Aux_S16 += t;
   }
   else {
      Aux_S16 -= t;
   }
   return Aux_S16;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	312	308	308	312	320	312	308	308	312	320	308	308	308
Conditional Branches	7	6	6	6	6	7	6	6	6	6	6	6	6
Unconditional Branches	3	3	3	3	3	3	3	3	3	3	3	3	3
Number of NOPs	5	4	4	4	4	5	4	4	4	4	4	4	4
Number of MBBs	17	15	15	15	15	17	15	15	15	15	16	15	15
Max Cycles	124	125	125	126	128	124	124	124	126	128	125	125	125
Min Cycles	51	52	52	53	55	51	51	51	53	55	52	52	52
Deviation	0.5887	0.584	0.584	0.5794	0.5703	0.5887	0.5887	0.5887	0.5794	0.5703	0.584	0.584	0.584
Number of movCCs	-	1	-	-	-	-	1	-	-	-	1	-	-
Number of selCCs	-	-	1	-	-	-	-	1	-	-	-	1	1
Number of predbegins	-	-	-	1	1	-	-	-	1	1	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.13 Interpolation Table – Single-Path

Although the single-path variant of the interpolation table algorithm shows worse performance on the SPARC V8 target, it strongly benefits from the instruction set extensions: If conditional select and hardware loop instructions are provided, the resulting code size is nearly the same as the first implementation and only shows a slightly increased worst-case performance. Note that the number of conditional branches could be reduced by 50 %.

```
static int16_t tab_lookup_sp(const tab * map, int16_t x)
{
   uint8_t N = map->Nx;
   uint16_t Aux_U16, uint16_t Aux_U16_a;
   uint16_t t;

   const int16_t * x_table_safe, * x_table;
   const int16_t * z_table ;

   x_table_safe = (const int16_t *) map->x_table;
   z_table = (const int16_t *) map->z_table;

   int i, p = 0;
   x_table = x_table_safe+1;
   for(i = 1; i < N-1; i++)
   {
     if(x > *(x_table++)) p++;
   }
   int16_t x0,x1,v0,v1;

   x_table = x_table_safe + p;
   x0 = *(x_table++);
   x1 = *x_table;
   if(x > map->x_table[N-1])x = x1;
   if(x < map->x_table[0])  x = x0;

   z_table += p;
   v0 = *(z_table++);
   v1 = *z_table;

   Aux_U16 = (uint16_t) (((uint16_t) x) - ((uint16_t) x0));
   Aux_U16_a = (uint16_t) (((uint16_t) x1) - ((uint16_t) x0));

   ...

   if (v0 > v1) t=-t;
   v0 += t;

   return v0;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	368	340	324	376	408	368	344	328	380	412	344	328	328
Conditional Branches	8	4	4	4	4	8	3	3	3	3	3	3	3
Unconditional Branches	3	1	1	2	2	3	1	1	2	2	1	1	1
Number of NOPs	4	2	2	2	2	4	2	2	2	2	2	2	2
Number of MBBs	19	9	9	10	10	19	10	10	11	11	10	10	10
Max Cycles	204	183	171	218	244	204	157	145	192	218	157	145	145
Min Cycles	194	183	171	218	244	194	157	145	192	218	157	145	145
Deviation	0.049	0	0	0	0	0.049	0	0	0	0	0	0	0
Number of movCCs	-	4	-	-	-	-	4	-	-	-	4	-	-
Number of selCCs	-	-	4	-	-	-	-	4	-	-	-	4	4
Number of predbegins	-	-	-	5	5	-	-	-	5	5	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	-	-
Number of HWLoops	-	-	-	-	-	0	1	1	1	1	1	1	1

B.14 Matrix Sum

Similar to the Fourier discrete cosine transformation, the matrix sum algorithm is mainly dependent on arithmetic instructions. LLVM is not able to introduce any of the presented instruction set extensions such that the performance is identical on all targets.

```
static void matrix_sum( int** inputmatrix,
                        int* rowsums,
                        int* colsums,
                        int size) {
  int i, j;
  // init loop
  for (i = 0; i < size; i++) {
    // init rowsums[i] with 0
    rowsums[i] = 0;
    // init colsums[i] with 0
    colsums[i] = 0;
  }
  // sum up rows and columns
  for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
      rowsums[i] += matrix[i][j];
      colsums[j] += matrix[i][j];
    }
  }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	312	312	312	312	312	312	312	312	312	312	312	312	312
Conditional Branches	1	1	1	1	1	1	1	1	1	1	1	1	1
Unconditional Branches	1	1	1	1	1	1	1	1	1	1	1	1	1
Number of NOPs	1	1	1	1	1	1	1	1	1	1	1	1	1
Number of MBBs	4	4	4	4	4	4	4	4	4	4	4	4	4
Cycles	636	636	636	636	636	636	636	636	636	636	636	636	636
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	0	0	-	-	-	0	0	0	0	0
Nesting level of predicated blocks	-	-	-	-	-	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.15 Median with Quick Sort

The first version of finding the median element of a given array uses quick sort and returns the middle element of the sorted array. As can be seen in the following table, only predicated blocks may be introduced, but the resulting performance is nearly the same.

```
static elem_type median(elem_type m[], int n) {

  int start, end, lowerindex, upperindex;
  elem_type *tmp = m;
  elem_type element, tmpelement;
  start = 0; end = n - 1;
  if(!tmp)
    return ((elem_type) 0);

  while(1) {
    element = tmp[start];
    lowerindex = start + 1;
    upperindex = end;
    do {
      while (upperindex > (start + 1) && tmp[upperindex] >= element)
        upperindex--;

      while (lowerindex < end && tmp[lowerindex] <= element)
        lowerindex++;

      if (lowerindex < upperindex) {
        tmpelement = tmp[lowerindex];
        tmp[lowerindex] = tmp[upperindex];
        tmp[upperindex] = tmpelement;
      }

    } while (lowerindex < upperindex);

    if (tmp[upperindex] < element) {
      tmp[start] = tmp[upperindex];
      tmp[upperindex] = element;
    } else {
      upperindex = start;
    }

    if (upperindex < ((n + 1)/2 - 1)) {
      start = upperindex + 1;
    } else if (upperindex > ((n + 1)/2 - 1)) {
      end = upperindex - 1;
    } else {
      break;
    }
  }
  return tmp[(n+1)/2 - 1];
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	384	384	384	388	396	384	384	384	388	396	388	396	388
Conditional Branches	11	11	11	10	10	11	11	11	10	10	10	10	10
Unconditional Branches	7	7	7	7	7	7	7	7	7	7	7	7	7
Number of NOPs	10	10	10	10	10	10	10	10	10	10	10	10	10
Number of MBBs	28	28	28	26	26	28	28	28	26	26	26	26	26
Max Cycles	36712	36712	36712	36976	37042	36712	36712	36712	36976	37042	36976	37042	36976
Min Cycles	2783	2783	2783	2795	2805	2783	2783	2783	2795	2805	2795	2805	2795
Deviation	0.9242	0.9242	0.9242	0.9244	0.9243	0.9242	0.9242	0.9242	0.9244	0.9243	0.9244	0.9243	0.9244
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	1	1	-	-	-	1	1	1	1	1
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	0	-
Number of HWLoops	-	-	-	-	-	0	0	-	-	0	0	0	0

B.16 Median without Sorting

The second version of calculating the median value is based on the idea to guess the median in a first step and count the number of elements which are greater and which are less. If both values are less or equal to half of the number of array elements, the median has been found, otherwise, a new maximum or minimum value for the next guess is set. As can be seen in the following table, this solution is more efficient than the version using quick sort. Nevertheless, when introducing predicated blocks based on predicate registers, the performance decreases dramatically.

```
static elem_type torben(elem_type m[], int n) {

  int i, less, greater, equal;
  elem_type min, max, guess, maxltguess, mingtguess;
  min = max = m[0];

  for (i=1 ; i<n ; i++) {
    if (m[i]<min) min=m[i]; /* ai: flow (here) <= 64 "torben"; */
    if (m[i]>max) max=m[i]; /* ai: flow (here) <= 64 "torben"; */
  }

  while (1) {
    guess = (min+max)/2;
    less = 0; greater = 0; equal = 0;
    maxltguess = min;
    mingtguess = max;

    for (i=0; i<n; i++) {
      if (m[i]<guess) {
        less++;
        if (m[i]>maxltguess) maxltguess = m[i];
      } else if (m[i]>guess) {
        greater++;
        if (m[i]<mingtguess) mingtguess = m[i];
      } else equal++;
    }
    if (less <= (n+1)/2 && greater <= (n+1)/2) break;
    else if (less>greater) max = maxltguess;
    else min = mingtguess;
  }

  if (less >= (n+1)/2) return maxltguess;
  else if (less+equal >= (n+1)/2) return guess;
  else return mingtguess;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	396	368	364	416	472	396	372	368	424	484	388	420	384
Conditional Branches	15	12	12	10	7	15	11	11	9	5	9	5	9
Unconditional Branches	4	4	4	4	2	4	4	4	4	2	4	2	4
Number of NOPs	11	10	10	11	6	11	9	9	11	6	11	6	1
Number of MBBs	29	23	23	21	14	29	24	24	22	16	22	16	22
Max Cycles	29309	29018	29017	36226	68208	29309	28830	28829	36102	62354	35652	61645	35651
Min Cycles	2030	1708	1707	2094	3654	2030	1520	1519	1970	3339	1520	2630	1519
Deviation	0.9307	0.9411	0.9412	0.9422	0.9464	0.9307	0.9473	0.9473	0.9454	0.9465	0.9574	0.9573	0.9574
Number of movCCs	-	3	-	-	-	-	3	-	-	-	3	-	-
Number of selCCs	-	-	3	-	-	-	-	3	-	-	-	3	3
Number of predbegins	-	-	-	5	12	-	-	-	5	12	2	8	2
Nesting level of predicated blocks	-	-	-	-	2	-	-	-	-	2	-	2	-
Number of HWLoops	-	-	-	-	-	0	1	1	1	2	1	2	1

B.17 Median without Sorting – Single-Path

The single-path variant is based on the previous version of the median algorithm and includes knowing the upper bound of the outer loop. The code size and worst-case performance of the SPARC V8 target is much worse than for the first two versions. However, targets implementing conditional move or select instructions and support hardware loops show a quite good worst-case performance and have a reduced number of conditional branches.

```
static elem_type torben_sp(elem_type m[], int n) {

  int i, less, greater, equal;
  elem_type min, max, guess, maxltguess, mingtguess;
  elem_type tmp, med; /* defined for sp-conversion */
  int j, boolean; /* defined for sp-conversion */
  min = max = m[0];

  for (i = 1 ; i < n ; i++) {
    tmp = m[i];
    if (tmp < min) min = tmp;
    if (tmp > max) max = tmp;
  }

  for (i = 0; i < ((NUMBER_OF_ELEMENTS + 1)/2); i++) {
    guess = (min+max)/2;
    less = 0; greater = 0; equal = 0;
    maxltguess = min;
    mingtguess = max;

    for (j=0; j < n; j++) {
      tmp = m[j];
      if (tmp < guess) less++;
      if (tmp < guess && tmp > maxltguess) maxltguess = tmp;
      if (tmp > guess) greater++;
      if (tmp > guess && tmp < mingtguess) mingtguess = tmp;
      if (tmp == guess) equal++;
    }

    if ((less <= (n+1)/2) && (greater <= (n+1)/2)) boolean = 0;
    if (boolean && (less > greater)) max = maxltguess;
    if (boolean && (less <= greater)) min = mingtguess;
  }

  med = mingtguess;
  if (less >= (n+1)/2) med = maxltguess;
  if ((less + equal >= (n+1)/2) && (less < (n+1)/2)) med = guess;
  return med;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	640	504	428	688	848	640	512	436	696	856	512	436	436
Conditional Branches	25	5	5	5	5	25	3	3	4	4	3	3	3
Unconditional Branches	3	3	3	3	3	3	3	3	3	3	3	3	3
Number of NOPs	12	4	4	4	4	12	3	3	4	4	3	3	3
Number of MBBs	52	12	12	12	12	52	14	14	13	13	14	14	14
Max Cycles	96935	70808	57606	110136	140888	96935	64317	51115	110012	140764	64317	51115	51115
Min Cycles	92581	70808	57606	110136	140888	92581	64317	51115	110012	140764	64317	51115	51115
Deviation	0.0449	0	0	0	0	0.0449	0	0	0	0	0	0	0
Number of movCCs	-	20	-	-	-	-	20	-	-	-	20	-	-
Number of selCCs	-	-	20	-	-	-	-	20	-	-	-	20	20
Number of predbegins	-	-	-	20	20	-	-	-	20	20	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	-	-
Number of HWLoops	-	-	-	-	-	0	2	2	1	1	2	2	2

B.18 Shell Sort

The shell sort algorithm is based on insertion sort, but divides the array into small parts in a first step.[1] As can be seen in the following table, none of the proposed instruction set extensions could be introduced.

```
static void sort(AbstractType *data, int N)
{
  // using shellsort with Ciura Sequence
  // http://en.wikipedia.org/wiki/Shell_sort
  // http://sun.aei.polsl.pl/~mciura/publikacje/shellsort.pdf
  int i, j, k, h;
  AbstractType v;
  //Input length is limited, so we don't need to use all increments
  const int incs[2] =   {4,1};
  for ( k = 0; k < 2; k++) {
    for (h = incs[k], i = h; i <N; i++)/*ai: loop here MAX 6 */
    {
      v = data[i];
      j = i;
      while ((j >= h) && data[j-h] > v) /*ai: loop here MAX 5 */
      {
        data[j] = data[j-h];
        j -= h;
      }
      data[j] = v;
    }
  }
}
```

[1] For a detailed description see `http://en.wikipedia.org/wiki/Shellsort`, accessed 2012-02-10.

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	388	388	388	388	388	388	388	388	388	388	388	388	388
Conditional Branches	10	10	10	10	10	10	10	10	10	10	10	10	10
Unconditional Branches	7	7	7	7	7	7	7	7	7	7	7	7	7
Number of NOPs	6	6	6	6	6	6	6	6	6	6	6	6	6
Number of MBBs	22	22	22	22	22	22	22	22	22	22	22	22	22
Max Cycles	312	312	312	312	312	312	312	312	312	312	312	312	312
Min Cycles	164	164	164	164	164	164	164	164	164	164	164	164	164
Deviation	0.4744	0.4744	0.474	0.4744	0.4744	0.4744	0.4744	0.4744	0.4744	0.4744	0.4744	0.4744	0.4744
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	0	0	-	-	-	0	0	0	0	0
Nesting level of predicated blocks	-	-	-	-	-	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.19 Shell Sort – Single-Path

The single-path variant of shell sort is the single algorithm of the current evaluation, which shows constant execution time only if nested if-then-else elimination is supported. However, the resulting performance is only competitive if the target also supports conditional select instructions as can be seen in the following table.

```
static void sort(AbstractType *data, int N)
{
  int i, j, k, h,lastj;
  bool abort,done;
  AbstractType v;
  //Input length is limited, so we don't need to use all increments
  const int incs[2] =   {4,1};
  N=6;
  for ( k = 0; k < 2; k++)
  {
    for (h = incs[k], i = h; i < 6; i++)
    {
      v = data[i];
      j = i;
      lastj = i;

      abort = (data[j-h] < v); //initial loop conditions
      done = false;

      for( ; j >= h; j -= h)
      {
        if(!done && !abort) abort = (data[j-h] < v);
        if(!done && !abort) data[j] = data[j-h];
        if(!done && abort) done = true;
        if(done && abort) lastj = j;
        if(done && abort) abort = false;
      }
      if(!done) lastj = j; //loop terminated without a break
      data[lastj] = v;
    }
  }
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	640	620	532	652	748	640	620	532	652	748	620	552	532
Conditional Branches	20	10	10	9	8	20	10	10	9	8	9	8	9
Unconditional Branches	4	3	2	2	2	4	3	2	2	2	3	2	2
Number of NOPs	12	5	5	4	4	12	5	5	4	4	4	4	4
Number of MBBs	41	22	21	19	17	41	22	21	19	17	20	17	19
Max Cycles	1255	1267	1004	1443	1813	1255	1267	1004	1443	1813	1309	1147	1046
Min Cycles	1132	1129	892	1339	1813	1132	1129	892	1339	1813	1231	1147	994
Deviation	0.0980	0.1089	0.1116	0.0721	0	0.0980	0.1089	0.1116	0.0721	0	0.0596	0	0.04971
Number of movCCs	-	10	-	-	-	-	10	-	-	-	10	-	-
Number of selCCs	-	-	10	-	-	-	-	10	-	-	-	10	10
Number of predbegins	-	-	-	13	14	-	-	-	13	14	1	2	1
Nesting level of predicated blocks	-	-	-	-	1	-	-	-	-	1	-	0	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.20 Software Division – Naive Implementation

The first version of the software division algorithm simply subtracts the divisor from the dividend, saves the result to the dividend and increments a counter. This is done as long as the dividend is greater than the divisor. The counter value is the result of the division. The covered test cases divided 65 535 by all possible 16-bit values from 0 to 65 535. As can be seen in the following table, the difference between maximum and minimum cycle count is tremendous.

```
static uint16_t divide_simple(uint16_t a, uint16_t b) {
  uint16_t result = 0;
  if (b == 0) {
    return UINT16_MAX;
  }
  while ((int32_t) (a - b) >= 0) {
    a = a - b;
    result++;
  }
  return result;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	224	224	224	224	224	224	224	224	224	224	224	224	224
Conditional Branches	7	7	7	7	7	7	7	7	7	7	7	7	7
Unconditional Branches	1	1	1	1	1	1	1	1	1	1	1	1	1
Number of NOPs	6	6	6	6	6	6	6	6	6	6	6	6	6
Number of MBBs	13	13	13	13	13	13	13	13	13	13	13	13	13
Max Cycles	458753	458753	458753	458753	458753	458753	458753	458753	458753	458753	458753	458753	458753
Min Cycles	4	4	4	4	4	4	4	4	4	4	4	4	4
Deviation	1	1	1	1	1	1	1	1	1	1	1	1	1
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	0	0	-	-	-	0	0	0	0	0
Nesting level of predicated blocks	-	-	-	-	-	-	-	-	-	-	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.21 Software Division – Shift Implementation

The second variant of the software division algorithm is based on the manual division as usually learned in school for decimal numbers: It is based on shifting the divisor to the left and subtract it from the dividend. Although the resulting deviation is still near 1, the worst case cycle count could be reduced from over 4 million to only a few hundred cycles, as can be seen in the following table.

```
static uint16_t divide_shift(uint16_t a, uint16_t b) {
  uint16_t result = 0;
  uint16_t divisor = b;

  if (b == 0) {
    return UINT16_MAX;
  }

  while (b < a && !(b & (uint16_t) (1<<15))) {
    b = b << 1;
  }

  while (b >= divisor) {
    result = result << 1;
    if (a >= b) {
      result = result + 1;
      a = a - b;
    }
    b = b >> 1;
  }
  return result;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	352	332	324	356	372	352	332	324	356	372	332	324	324
Conditional Branches	12	10	10	10	10	12	10	10	10	10	10	10	10
Unconditional Branches	2	1	2	1	1	2	1	2	1	1	1	2	2
Number of NOPs	7	6	6	6	6	7	6	6	6	6	6	6	6
Number of MBBs	20	15	16	15	15	20	15	16	15	15	15	16	16
Max Cycles	398	350	314	446	510	398	350	314	446	510	350	314	314
Min Cycles	4	4	4	4	4	4	4	4	4	4	4	4	4
Deviation	0.9899	0.9886	0.9873	0.9910	0.9922	0.9899	0.9886	0.9873	0.9910	0.9922	0.9886	0.9873	0.9873
Number of movCCs	-	2	-	-	-	-	2	-	-	-	2	-	-
Number of selCCs	-	-	2	-	-	-	-	2	-	-	-	2	2
Number of predbegins	-	-	-	3	3	-	-	-	3	3	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	-	-
Number of HWLoops	-	-	-	-	-	0	0	0	0	0	0	0	0

B.22 Software Division – Single-Path

The last version is a single-path transformation of the software division algorithm. Although it shows a larger code size, the worst case performance could still be reduced when conditional move or select as well as loop instructions are supported. Moreover, the number of conditional branches has decreased about 50 % in comparison to the shift implementation as can be seen in the following table.

```
static uint16_t divide_wcet(uint16_t a, uint16_t b) {
  int i = 0;
  int boolean1;
  int boolean2;

  uint16_t result = 0;
  uint16_t divisor = b;

  for (i = 0; i < 16; i++) {
    boolean1 = (b & (uint16_t) (1 << 15));
    b = boolean1 ? b : b << 1;
  }

  for (i = 0; i < 16; i++) {
    boolean1 = (b >= divisor);
    result = boolean1 ? result << 1 : result;
    boolean2 = (a >= b);
    result = (boolean1 && boolean2) ? result + 1 : result;
    a = (boolean1 && boolean2) ? a - b : a;
    b = boolean1 ? b >> 1 : b;
  }

  boolean1 = (b != 0);
  result = boolean1 ? result : UINT16_MAX;
  return result;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	556	532	516	564	596	556	536	520	568	600	536	520	520
Conditional Branches	10	6	6	6	6	10	5	5	5	5	5	5	5
Unconditional Branches	1	1	1	1	1	1	1	1	1	1	1	1	1
Number of NOPs	7	5	5	5	5	7	5	5	5	5	5	5	5
Number of MBBs	20	12	12	12	12	20	13	13	13	13	13	13	13
Max Cycles	475	412	346	495	593	475	368	302	451	549	368	302	302
Min Cycles	429	412	346	495	593	429	368	302	451	549	368	302	302
Deviation	0.1133	0	0	0	0	0.1133	0	0	0	0	0	0	0
Number of movCCs	-	4	-	-	-	-	4	-	-	-	4	-	-
Number of selCCs	-	-	4	-	-	-	-	4	-	-	-	4	4
Number of predbegins	-	-	-	4	4	-	-	-	4	4	0	0	0
Nesting level of predicated blocks	-	-	-	-	0	-	-	-	-	0	-	-	-
Number of HWLoops	-	-	-	-	-	0	1	1	1	1	1	1	1

B.23 Threshold

The threshold algorithm checks for every element of an array, whether it is within the given bounds, i.e., a maximum and a minimum level. As can be seen in the following table, the cycle count is equal for all targets, although some of the instruction set extensions have been used. This is due to the fact that only occur within the main and initialization functions and thus do not influence the cycle count of the threshold algorithm itself.

```
static int threshold(int* array, int size, int min_val, int max_val) {
  int i;
  int counter = 0;
  for (i = 0; i < size; i++) {
    if (array[i] < min_val) {
      array[i] = min_val;
      counter++;
      continue;
    } else if (array[i] > max_val ) {
      array[i] = max_val;
      counter++;
      continue;
    }
  }
  return counter;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	500	500	500	504	540	500	500	500	504	540	504	540	504
Conditional Branches	13	13	13	12	9	13	13	13	12	9	12	9	12
Unconditional Branches	6	6	6	6	4	6	6	6	6	4	6	4	6
Number of NOPs	8	8	8	8	5	8	8	8	8	5	8	5	8
Number of MBBs	28	28	28	27	21	28	28	28	27	21	27	21	27
Max Cycles	23555	23555	23555	23555	23555	23555	23555	23555	23555	23555	23555	23555	23555
Min Cycles	20483	20483	20483	20483	20483	20483	20483	20483	20483	20483	20483	20483	20483
Deviation	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304	0.1304
Number of movCCs	-	0	-	-	-	-	0	-	-	-	0	-	-
Number of selCCs	-	-	0	-	-	-	-	0	-	-	-	0	0
Number of predbegins	-	-	-	1	6	-	-	-	1	6	1	6	1
Nesting level of predicated blocks	-	-	-	-	3	-	-	-	-	3	-	3	-
Number of HWLoops	-	-	-	-	-	0	0	-	0	0	0	0	0

B.24 Threshold – Single-Path

The single-path implementation of the threshold algorithm shows a worse performance on the SPARC V8 target, but benefits a lot from conditional move and select instructions if they are introduced in combination with hardware loops: The code size as well as the worst-case cycle count is less than for the first implementation and the number of conditional branches has decreased. Like before, some of the introduced instruction set extensions occur in the main and initialization function and do not influence the cycle count of the algorithm.

```
static int threshold(int* array, int size, int min_val, int max_val) {
  int i;
  int counter = 0;
  int tmp;
  for (i = 0; i < size; i++) {
    tmp = array[i];
    if (tmp < min_val) {
      tmp = min_val;
      counter++;
    }
    if (array[i] > max_val ) {
      tmp = max_val;
      counter++;
    }
    array[i] = tmp;
  }
  return counter;
}
```

	v8	**v8-m**	**v8-s**	**v8-i**	**v8-r**	**v8-l**	**v8-ml**	**v8-sl**	**v8-il**	**v8-rl**	**v8-mil**	**v8-srl**	**v8-sil**
Code Size	532	504	484	540	608	532	508	488	544	612	512	528	492
Conditional Branches	15	11	11	10	7	15	10	10	9	6	9	6	9
Unconditional Branches	5	5	5	5	3	5	5	5	5	3	5	3	5
Number of NOPs	9	6	6	6	3	9	6	6	6	3	6	3	6
Number of MBBs	31	23	23	22	16	31	24	24	23	17	23	17	23
Max Cycles	35844	27652	24579	36868	45060	35844	24584	21511	33800	41992	24584	21511	21511
Min Cycles	33796	27652	24579	36868	45060	33796	24584	21511	33800	41992	24584	21511	21511
Deviation	0.0571	0	0	0	0	0.0571	0	0	0	0	0	0	0
Number of movCCs	-	4	-	-	-	-	4	-	-	-	4	-	-
Number of selCCs	-	-	4	-	-	-	-	4	-	-	-	4	4
Number of predbegins	-	-	-	5	10	-	-	-	5	10	1	6	1
Nesting level of predicated blocks	-	-	-	-	3	-	-	-	-	3	-	3	-
Number of HWLoops	-	-	-	-	-	0	1	1	1	1	1	1	1

Bibliography

[Ana08] Analog Devices, Norwood, Mass. *Blackfin® Processor Programming Reference*, September 2008. Revision 1.3.

[Ana10] Analog Devices, Norwood, Mass. *ADSP-BF50x Blackfin® Processor Hardware Reference*, December 2010. Revision 1.0.

[Ana11] Blackfin processor homepage, 2011. URL `http://www.analog.com/en/processors-dsp/blackfin/products/index.html`. Accessed: 2011-08-24.

[ARM05] ARM, Cambridge, England. *ARM Architecture Reference Manual*, July 2005.

[ARM09] ARM, Cambridge, England. *ARM1136JF-S™ and ARM1136J-S™ Technical Reference Manual*, February 2009. Revision r1p5.

[ARM10] ARM, Cambridge, England. *ARM®v7-M Architecture Reference Manual*, November 2010. Errata markup.

[ARM11a] Arm processor homepage, 2011. URL `http://www.arm.com/products/processors/index.php`. Accessed: 2011-08-30.

[ARM11b] ARM, Cambridge, England. *ARM® Architecture Reference Manual – ARM®v7-A and ARM®v7-R edition*, July 2011. Errata markup.

[ASP+03] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg, and Frank Mueller. Virtual simple architecture (visa): Exceeding the complexity limit in safe real-time systems. In *In International Symposium on Computer Architecture*, pages 250–261. IEEE Computer Society, 2003.

[Atm10] Atmel, San Jose, CA. *8-bit AVR® Instruction Set*, July 2010.

[Atm11] Atmel, San Jose, CA. *AVR32 Architecture Document*, April 2011.

[CS91] Bryce Cogswell and Zray Segall. Macs: a predictable architecture for real time systems. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 296–305. IEEE Computer Society, December 1991.

[Del02] Martin Delvai. Handbuch für spear (scalable processor for embedded applications in real-time environments). Research Report 70/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[DHPS03] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability - the spear design example. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 169–176. IEEE Computer Society, July 2003.

[EL07] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 264–265. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-627-1.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319 – 349, July 1987.

[GK07] Woon-Seng Gan and Sen M. Kuo. *Embedded signal processing with the Micro Signal Architecture*, chapter 5 – Introduction to the Blackfin Processor, pages 163 – 216. John Wiley & Sons, Hoboken, Canada, 2007.

[Gon00] Ricardo E. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20(2):60–70, March/April 2000. ISSN 0272-1732.

[GS05] John Goodacre and Andrew N. Sloss. Parallelism and the arm instruction set architecture. *Computer*, 38(7):42 – 50, July 2005. ISSN 0018-9162.

[GTS07] Johann Großschädl, Stefan Tillich, and Alexander Szekely. Performance evaluation of instruction set extensions for long integer modular arithmetic on a sparc v8 processor. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 680–689, August 2007.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. ISSN 0018-9219.

[IE06] Nicholas Ip and Stephen Edwards. A processor extension for cycle-accurate real-time software. In Edwin Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon-Hae Kim, Laurence Yang, and Bin Yiao, editors, *Embedded and Ubiquitous Computing*, volume 4096 of *Lecture Notes in Computer Science*, pages 449–458. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-36679-9.

[Inf03a] Infineon, München, Germany. *TriCore™ 32-bit Unified Processor DSP Optimization Guide, Part 1: Instruction Set*, January 2003. Version 1.6.4.

[Inf03b] Infineon, München, Germany. *TriCore™ Compiler Writer's Guide*, December 2003. Version 1.4.

[Kop97] Hermann Kopetz. *Real-time Systems. Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[KP05] Raimund Kirner and Peter Puschner. Classification of wcet analysis techniques. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE Computer Society Press, Washington DC, 2005.

[KP11] Raimund Kirner and Peter Puschner. Time-predictable computing. In Sang Minand Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 23–34. Springer Berlin / Heidelberg, 2011.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, March 2004.

[Lat11] Chris Lattner. Llvm. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications. Elegance, Evolution and a Few Fearless Hacks*, chapter 11. lulu.com, June 2011. ISBN 978-1-257-63801-7. URL `http://www.aosabook.org/en/index.html`. Accessed: 2012-01-16.

[LLK+08] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 137–146. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-469-0.

[LLV11] The llvm compiler infrastructure project homepage, 2011. URL `http://llvm.org/`. Accessed: 2011-10-23.

[LRL10] Isaac Liu, Jan Reineke, and Edware A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pages 2111–2115, November 2010. ISSN 1058-6393.

[LSMA99] Lea Hwang Lee, Jeff Scott, Bill Moyer, and JOhn Arends. Low-cost branch folding for embedded applications with small tight loops. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 103–111, 1999.

[Mal11] Mälardalen wcet project / benchmarks, 2011. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. Accessed: 2011-10-02.

[MHM+95] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen mei W. Hwu. A comparison of full and partial predicated execution support

for ilp processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 138–149, June 1995. ISSN 1063-6897.

[PB00] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18:115–128, 2000. ISSN 0922-6443.

[PB02] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02)*. IEEE Computer Society Press, 2002.

[PS91] Joseph C.H. Park and Mike Schlansker. On predicated execution. Technical report, Hewlett Peckard Software and Systems Laboratory, May 1991.

[Pus03] Peter Puschner. The single-path approach towards wcet-analysable software. In *IEEE International Conference on Industrial Technology (ICIT'03)*. Tiskarna tehniških fakultet, Maribor, 2003.

[Pus07] Peter Puschner. Evaluation of the single-path approach and wcet-oriented programming. Research report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, January 2007.

[Sch05a] Martin Schöberl. Evaluation of a java processor. In *Tagungsband Austrochip 2005*, pages 127–134. Vienna, Austria, October 2005.

[Sch05b] Martin Schöberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Technische Universität Wien, Vienna, Austria, January 2005.

[Sch09a] Martin Schöberl. *JOP Reference Handbook*. CreateSpace, 2009. ISBN 978-1438239699.

[Sch09b] Martin Schöberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009.

[Sch11] Martin Schöberl. Jop homepage, 2011. URL `http://www.jopdesign.com/`. Accessed: 2011-10-16.

[SPA92] SPARC International, Menlo Park, CA. *The SPARC Architecture Manual – Version 8*, 1992.

[SSP+11] Martin Schöberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–21, 2011.

[Sta90] Johan A. Stankovic. The spring architecture. In *Real Time, 1990. Proceedings., Euromicro '90 Workshop on*, pages 104–113. IEEE Computer Society Press, June 1990.

[Ten02] Tensilica Inc., Santa Clara, CA, USA. *Xtensa® Microprocessor Overview Handbook*, August 2002.

[Ten10] Tensilica Inc., Santa Clara, CA, USA. *Xtensa® Instruction Set Architecture (ISA) Reference Manual*, September 2010.

[TG06] Stefan Tillich and Johann Großschädl. Instruction set extensions for efficient aes implementation on 32-bit processors. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 270–284. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-46559-1.

[TW04] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157–177, 2004. ISSN 0922-6443.

[VATJ06] Brian F. Veale, John K. Antonio, Monte P. Tull, and Sean A. Jones. Selection of instruction set extensions for an fpga embedded processor core. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[WA06] Jack Whitham and Neil C. Audsley. Mcgrep–a predictable architecture for embedded real-time systems. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 13–24. IEEE Computer Society, December 2006. ISSN 1052-8725.

[WA08] Jack Whitham and Neil C. Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 305–316. IEEE Computer Society, April 2008. ISSN 1080-1812.

[YM05] Pan Yu and Tulika Mitra. Satisfying real-time constraints with custom instructions. In *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, pages 166–171, September 2005.

[YZ08] Jun Yan and Wei Zhang. A time-predictable vliw processor and its compiler support. *Real-Time Systems*, 38:67–84, January 2008.

[Zha97] Lichen Zhang. Predictable architecture for real-time systems. In *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, volume 3, pages 1761–1765. IEEE Press, September 1997.

Index