Time-Predictable Code Execution — Instruction-Set Support for the Single-Path Approach

Clemens B. Geyer, Benedikt Huber, Daniel Prokesch, and Peter Puschner
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
clemens.geyer@gmail.com
{benedikt,daniel,peter}@vmars.tuwien.ac.at

***Abstract*—When designing modern real-time systems, which have to deliver results at specified deadlines, knowing the worst-case execution time (WCET) of software components is of utmost importance. Although there has been much research in the field of WCET analysis in the last years, with a focus on improving the accuracy of processor models and WCET-calculation methods, researchers have paid little attention to exploring the impact of the instruction set architecture (ISA) on the time predictability of the code executing on a given real-time processor. In this paper we explore ISA extensions that allow compilers to generate highly time-predictable code. To this end, an existing instruction set has been extended by a number of instructions, and the LLVM compiler framework has been adapted to use these new instructions in its assembler-code generator. The timing behavior of the generated code has been evaluated by means of an instruction-set simulator. The results of the experiments allowed us to identify a promising combination of the newly introduced instructions. The use of these instructions leads to a reduction of the number of branches in the assembler code, thus improving time predictability while still providing competitive worst-case timing.**

I. Introduction

Real-time systems are widely in use in the automotive and aerospace industries. In contrast to desktop and server applications, real-time systems of those domains have to provide their service reliably, for the full duration of their mission. Further, they have to deliver results on time. Otherwise a catastrophe like a plane crash might occur. Thus, an important aspect when dealing with real-time systems is the timing predictability of the implemented algorithms. *Worst-case scenarios* have to be identified such that the timely operation of the real-time computer system can be guaranteed for all phases of the system's operation.

Usually, 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 [1]. 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, and thus often delivers pessimistic results.

So far the research community has paid little attention to investigate the instruction-set architectures of processors with respect to how specific instructions can help to improve the overall timing predictability of real-time computer systems. Our hypothesis is, however, that the provision of a few special instructions can help compilers to generate time-predictable code. For example, a processor that offers the possibility to conditionally execute an instruction, usually denoted as a *predicated instruction*, allows the compiler to replace the simple branching code generated from if-then-else structures by strictly sequential blocks of the same functionality. In [2], Puschner and Burns build on this idea to generate fully time-predictable code for processors that provide a fully or partially predicated instruction set. Their so-called *single-path* code-generation strategy uses the predicated instructions of the processor to generate code with a single, predetermined control-flow path, thus eliminating one of the main sources of timing unpredictability at system-construction time.

In this paper we explore how the time-predictable execution of single-path code can benefit from specific machine instructions or instruction-set extensions. To this end we have added instructions to the SPARC V8 instruction set (like predicated instructions and loop instructions) and generated single-path code based on these modified instruction sets. The time predictability, worst-case execution time and code size of the generated code have been evaluated for a number of program examples that have been compiled for the different ISA variants. The contributions of the paper are as follows:

- We show the impact of the coding style in a high-level programming language on the code generation process – the WCET and timing predictability of the generated code – for different instruction-set architectures.
- We identify outstanding combinations of instruction-set extensions that provide high time predictability, facilitate good worst-case performance, and cause only a small increase in code size.

The paper is organized as follows: We first introduce the ideas behind the single-path code generation. Following this, Section III analyses instruction sets of existing processors and time-predictable architectures to identify promising instruction-set extensions. In Section IV we introduce the extensions of the SPARC V8 instruction set [3] that we consider relevant for the purpose of this paper. These instructions have been implemented in an instruction set simulator for the SPARC V8 ISA, and a the LLVM compiler framework has been adapted to generate code using the new ISA extensions. Section V summarizes the results of our evaluations in which we assessed the impact of the programming style on the generated code as well as WCET, time predictability and number of generated branches for the different ISA extensions.

II. Single-Path Code Generation

Our strategy for time-predictable code generation has its roots in the observation that input-dependent branches in the control flow of code may yield different instruction and memory-access traces at runtime. These different traces and memory access patterns may in turn cause variations in execution times on repeated executions of the code. To avoid these control-flow dependent timing variations, the *single-path code generation* [2] emits code that forces all runs of a program to execute the same trace of instructions.

The objective of the single-path translation is to remove input-data dependencies in the control flow. To achieve this, the single-path translation replaces all input-data dependent branching operations in the code by predicated code or other constructs that do not bring about input-dependent branches in the control flow. It serializes the input-dependent alternatives of the code and uses predicates or flags (instead of branches) and, if necessary, speculative execution to select the right code to be executed at runtime. All loops with an input-data dependent termination condition are transformed into loops with a constant number of iterations. In this loop conversion the termination condition of the original loop is transformed into a condition that occurs in the body of the new loop and makes the loop body execute conditionally, thus simulating the execution semantics of the original loop. Details about the single-path code generation, including code-generation rules, can be found in [4].

A. WCET-Oriented Programming

A traditional compiler inserts branch instructions into the code to ensure that alternative code executes mutually exclusive, i.e., in the case of an if-then-else construct only the instructions of one of the alternatives will be executed; the other alternative is excluded from execution. In contrast, the trace of a single-path program includes the instructions of all input-data dependent alternatives (though only a subset of the instructions affects the state of the variables and registers), which implies that the execution times of the different alternatives sum up. So if a program includes many input-data dependent alternatives then a worst-case execution of a single-path program might consume much more time than its traditional counterpart.

To avoid that single-path code generation yields code with a poor worst-case performance, software developers should adopt a programming style that avoids input-data dependent alternatives. Similarly, they should use algorithms and libraries that have been coded with a focus on an input-data independent control flow. We proposed such a programming strategy, called *WCET-Oriented Programming* [5], in earlier work.

WCET-oriented programming tries to treat all inputs equally. It avoids those traditional performance optimizations that generate input-data dependent branches to favor frequent execution scenarios of an application over rare ones. To this end, WCET-oriented programming tries to implement or use algorithms that are free from input-data dependent control-flow decisions or – if this cannot be completely achieved – to restrict the number of operations being only executed for a subset of the input-data space to a minimum.

WCET-oriented programming cannot be automated, as the development of algorithms is a creative act. However, WCET-oriented programming can be assisted by providing code examples and libraries, or tool support to identify input-data dependencies in code, e.g., a programming editor that highlights input-data dependent conditions and sections of code while the code is being programmed [6].

III. Analysis of Existing Processors

In order to identify instruction set extensions which might be favorable for hard real-time systems, we compare existing hardware architectures from the embedded system and real-time domain. We briefly characterize the selected commercial and academic architectures, focusing on properties which seem most relevant to our discussion. In particular, we pay attention to features that are favorable for the single-path approach, or benefit from WCET-oriented single-path code.

A. ARM processor family

One of the main objectives guiding the design of ARM processors is low power consumption, which motivated the choice of a simple RISC instruction set. In order to eliminate branches, the (standard) ARM instruction set is *fully predicated*. That is, for (almost) every instruction there are variants which are only executed if certain flags in the processors status register are set. The status register itself is modified by arithmetic and compare operations.

In order to increase code density, ARM processors feature another instruction set, called *Thumb instruction set*. This instruction set covers the most common instructions, encoded using 16 instead of 32 bits. The Thumb instruction set does not have enough bits to provide predication for all instructions in a direct way. It still supports fully predicated execution by means of the `IT` instruction, which controls whether the four subsequent instructions will be disabled. This innovative instruction inspired one of the instruction set extension presented in the next section.

The ARM instructions set also includes SIMD (single instruction multiple data) instructions, which provide a form of statically scheduled, and thus predictable parallelism. For a comprehensive description of the ARM instruction set, refer to [7].

B. Blackfin

The Blackfin processor by Analog Devices is a 32-bit RISC architecture and is a typical example of an embedded processor that includes features of digital signal processors (DSPs) [8]. Besides a hardware implementation of a minimum and maximum function and several SIMD instructions, the Blackfin processor supports hardware loops. The `lsetup` instructions takes three parameters: the starting and end address of the loop as well as the value for the number of loop iterations. It is also possible to implement nested hardware loops, as there are special loop registers with different priorities. The hardware loop instruction is relevant for our discussion, as it benefits from single-path code. Given that the initial value for the loop counter is input-data independent, it is certainly possible to use hardware loop instructions. Therefore, a variant of the `lsetup` instruction is part of the instruction set extensions considered in this article.

C. Java Optimized Processor

The Java Optimized Processor (JOP) [9] implements the Java Virtual Machine (JVM) in hardware and is designed to simplify WCET analysis. At a lower level, JOP executes a time-predictable RISC instruction set (microcode instructions). The timing behavior of each microcode instruction is known in advance and in most cases independent of the processor state. In order to execute complex JVM bytecode instructions, the processor translates them to a sequence of microcode instructions, using a statically known translation scheme. The timing behavior of high-level bytecode instructions can be derived by means of an automated analysis [10]. Additionally, the processor features novel cache architectures (e.g., method cache, stack cache), which allow to decouple the timing analysis of instructions from the analysis of caches.

D. Precision Timed Machine

The Precision Timed Machine (PRET) [11] was designed to permit the static analysis of the processor's exact timing behavior, and introduce timing semantics at the assembler level. A first version of a PRET simulator implementing the SPARC V8 instruction set was presented in [12]. In September 2010, a soft-core implementation based on the ARM instruction set has been released.

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 access and delay instructions, all instructions are guaranteed finish within one pipeline cycle.

A key instruction of the PRET architecture is the so-called *deadline* instruction [13]. This instruction delays execution in a cycle-accurate way, and allows programmers to implement precisely timed actions. Basically, the instruction sets an internal countdown timer to a given value, and delays subsequent instructions until the timer reaches zero. Despite its simplicity, the deadline instruction might provide new possibilities to realize systems with predictable timing.

E. Customizable Processors

The Tensilica Xtensa is an example of a customizable soft-core architecture. It provides the developer a basic set of features which may be fully adapted to personal needs [14]. The possible adaptions of features range from the individual scaling of the size of cache, RAM and ROM to the implementation of custom instructions. Additionally, the processor may be optimized for different goals like small die area or low power consumption.

The main application field of the Xtensa processor lies in hardware-software co-design: When features of an application-specific and a general purpose instruction set are needed, the Xtensa processor may be an adequate choice. Moreover, it offers the user the possibility to implement custom instructions which will be automatically added to the instruction set. Thus, the Xtensa processor is another interesting candidate to evaluate hardware implementations of promising instruction set extensions.

F. Summary

Processor / Microcontroller	Register Bit Width	Instruction Size (Bytes)	SIMD	Conditional Move/Select	Full Predication	Hardware Loops	Support for Single-Path Conversion
Blackfin	32	2 - 4[a]	yes	no	no	yes	−
ARM	32	2 - 4	yes	yes	yes	no	+
8-bit AVR	8	2 - 4	no	no	no	no	−
32-bit AVR	32	2 - 4	yes	yes	no	no	±
TriCore	32	2 - 4	yes	yes	no	no	±

[a] May be 8 bytes for a parallel instruction.

TABLE I. FEATURE OVERVIEW OF EVALUATED PROCESSOR ARCHITECTURES.

In addition to the architectures presented before, we analyzed the instruction set of 8-bit and 32-bit AVR processors and of the Infineon TriCore processor [15], two popular processor families for embedded systems. Table I gives an overview of the features of each evaluated commercial processor: Most processors implement a 32-bit architecture and provide SIMD instructions. However, only the ARM family provides a fully predicated instruction set, while most other processors provide conditional move or a select instruction. In theory, either is sufficient for single-path conversion, although the translation is more involved compared to a translation for full-predicated instruction sets [16].

The architectures analyzed in this section inspired the instruction set extensions presented next. In addition to predicated blocks and hardware loops, conditional move and conditional select instructions, which are the minimum support necessary to eliminate branches, will be investigated.

IV. Instruction Set Extensions

In order to evaluate the impact of several instruction set extensions, the SPARC V8 instruction set has been chosen for the following reasons:

- The SPARC V8 offers a general-purpose ISA with no special instructions like hardware loops.
- There are three unused opcodes that allow for the implementation of additional instructions.
- The LLVM compiler framework has a complete backend for the SPARC V8 and V9 instruction set which may easily be adapted to support new instructions.
- There is an open source soft-core implementation of SPARC V8 which can be used for evaluations on real hardware.

A. Syntax and Semantics of the Proposed Instruction Set Extensions

Figure 1 shows the syntax of the assembler code of the conditional move instruction: Based on the condition code **cc**, either the value of the source register *src* is copied to the destination register *dst* or the value of *dst* remains untouched. This instruction is the minimal requirement for a single-path conversion.

```
mov[cc] src, dst
```

Fig. 1. Conditional move instruction.

The conditional select instruction is shown in Figure 2. In contrast to the conditional move instruction, there are two source operands *src1* and *src2* and one destination register *dst*. If the condition code **cc** is satisfied, the value of the first source operand is copied to the destination register, otherwise the value of the second source operand. Thus, the conditional select instruction behaves like a hardware multiplexer.

```
sel[cc] src1, src2, dst
```

Fig. 2. Conditional select instruction.

In order to add support for a fully predicated instruction set without changing the encoding of every instruction, two extensions similar to the ARM *IT block* [7] have been developed. An ARM IT instruction allows up to four following instructions to be predicated. Our first implementation of *predicated blocks*, shown in Figure 3, is based on integer condition codes: All instructions following a `predbegin` instruction will be executed conditionally based on the condition code specified in the **cc** field, until either another `predbegin` or a `predend` instruction occurs.

```
predbegin[cc]
predend
```

Fig. 3. Predicated blocks based on integer condition codes

This implementation of predicated blocks has some limitations. Any instruction in the predicated block that changes flags in the status register might invalidate the condition of the executing block. As a result, converting nested if-then-else structures to single-path code would be inconvenient and the resulting code would be inefficient.

To overcome this limitation, we provide an alternative implementation of predicated blocks. It is based on predicate registers, which save whether the given condition is satisfied according to the current status register. Hence, changing any flags of the status register within a predicated block does not influence the previously saved condition. The predicate register file contains 16 predicate registers, each consisting of two flags.

Figure 4 shows the assembly code syntax. Predicate registers are three-valued: either the `t` flag or the `f` flag or none of them is set. The `predset` instruction sets the `t` and clears the `f` flag of the destination predicate register *rdp* in case the provided condition code **cc** is satisfied. Otherwise, the `f` flag is set and the `t` flag is cleared. The `predclear` instruction unconditionally clears both flags of *rdp*. This behavior is necessary for a simple conversion of nested if-then-else structures. In contrast to the first variant, the `predbegin` instruction has two operands: a source predicate register operand *rsp* and a 1-bit immediate operand **tf**, which can be either `t` or `f`. Only if the specified flag is set in *rsp*, the instructions of the predicated block are executed. Like for the first implementation of predicated blocks, the conditional execution of instructions – including `predset` and `predclear` – remains active until the next occurrence of a `predbegin` or `predend` instruction.

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

Fig. 4. Predicated blocks based on predicate registers.

The last proposed instruction set extension is a hardware loop instruction, which is particularly suitable for the efficient implementation of loops with a fixed iteration count. The syntax of our hardware loop extension, which is based on the corresponding instruction of the Blackfin processor, is depicted in Figure 5. A hardware loop uses three registers. One representing the start address of the loop (`%loops`), another one specifying the address of the first instruction after the loop (`%loope`), and a third one containing the number of loop iterations (`%loopb`). Moreover, there is a dedicated instruction to activate the hardware loop. Hardware loops may not appear to be relevant to our discussion at the first glance. However, the compiler will generate hardware loop instructions only if the number of loop iterations is statically known. In the case of single-path code, every loop has an input-data independent iteration count, and thus hardware loops are particularly effective.

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

Fig. 5. Instruction proposal for hardware loops.

B. Compiler Modifications

We adapted the LLVM SPARC backend to integrate the previously presented instruction set extensions into the assembly code generator.[1] Extending the code generator of the existing SPARC V8 to support conditional move and conditional select instructions was quite simple for 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 assembly syntax as the one we propose.
2) The LLVM intermediate representation facilitates a conditional select statement that can be easily mapped to a conditional move or conditional select instruction.

The integration of predicated blocks is based on the idea to replace conditional constructs by a straight-line single-path segment [17], as illustrated in Figure 6. If the branch condition is satisfied, block A is executed, block B otherwise. Thus, A becomes a predicated block with the original branch condition as predicate, B gets the negated branch condition as predicate. If the successor of A and B has no other predecessors, all four blocks may be merged (center). Otherwise, only the predecessor of A and both blocks A and B can be merged (right). By utilizing predicated blocks using predicate registers, this procedure can be applied recursively on the control-flow graph to convert nested if-then-else structures to single-path segments.

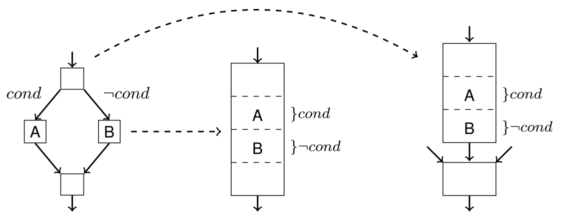

Fig. 6. Translation process of an if-then-else structure to assembler code and the resulting control flow graph. Each vertex corresponds to a machine basic block in the assembler code.

Figure 7 shows the compiler's output for a conditional branch implemented in C: In the corresponding assembler code without extensions enabled (labeled *SPARC V8*), there are four basic blocks, one conditional and one unconditional branch. If the compiler is configured to use predicated blocks, all branches are eliminated and the instructions corresponding to the conditional assignments in the source code are surrounded by `predbegin` and `predend` instructions. The code generated when conditional select instructions

[1]The source code of the adapted backend and installation instructions are available on GitHub (`https://github.com/cgeyer/llvm-cbg`).

C-Code:

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

SPARC V8:

```
branch_test:
  save %sp, -96, %sp
  subcc %i0, 5, %l0
  ble .LBB0_2
  nop
.LBB0_1:
  ba .LBB0_3
  or %g0, 3, %l0
.LBB0_2:
    or %g0, -7, %l0
.LBB0_3:
  smul %l0, %i0, %i0
  jmp %i7+8
  restore %g0, %g0, %g0
```

SPARC V8 + `selcc`:

```
branch_test:
  save %sp, -96, %sp
  subcc %i0, 5, %l0
  sel[g] 3, -7, %l0
  smul %l0, %i0, %i0
  jmp %i7+8
  restore %g0, %g0, %g0
```

SPARC V8 + `predblock`:

```
branch_test:
  save %sp, -96, %sp
  subcc %i0, 5, %l0
  predbegin[le]
  or %g0, -7, %l0
  predbegin[g]
  or %g0, 3, %l0
  predend
  smul %l0, %i0, %i0
  jmp %i7+8
  restore %g0, %g0, %g0
```

Fig. 7. The assembly code generated from a code snippet in C for a simple conditional branch (top left), using different approaches: the original SPARC V8 instruction set (bottom left), conditional selects (top right), predicated blocks (bottom right).

are enabled also consists of a single basic block. It is the most efficient solution in terms of code size and execution time for this example.

The hardware loop extension is utilized after the single-path conversion. At this point, most loops consist of a single basic block with a fixed iteration count, and so the algorithm outlined in Figure 8 is an effective optimization. A simple loop analysis inspects the basic block, and identifies the loop iteration register and the loop bound (line 4). Next, the algorithm inserts a new basic block before the original loop block to setup the hardware loop (line 5). In this block, the start and end address of the loop and its iteration count are set. Afterwards, the conditional branch and the compare instruction at the end of the loop are removed, because this is now done in hardware (line 6). Finally, all instructions that involve the loop iteration register and are no longer necessary are removed (line 7).

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

Fig. 8. Outline of the algorithm for utilizing hardware loops.

V. EVALUATION

We conducted several experiments to evaluate different combinations of the instruction set extensions presented before. First, we investigate the influence of algorithm design and coding style on the effectiveness of the instruction set extensions. We selected four algorithmic problems, and collected several implementations for each of them. The results for two of these problems are presented in Section V-A. Second, the influence of the implemented instruction set extensions on code size, the number of branches and the maximum execution time were evaluated, and are presented in Section V-B. For additional experimental results, please refer to [15].

All implementations were written in C, and translated to SPARC assembly using the extended LLVM backend presented in the previous section. Additionally, we implemented an instruction set simulator for the extended SPARC V8 ISA, and instrumented the programs to collect the number of processor cycles for each program.[2] The best- and worst case cycle count was approximated by taking the minimum and maximum over all input data vectors.

A. Impact of Coding Style on the Effectiveness of Instruction Set Extensions

In a first series of experiments, we simulated different implementations of selected algorithmic problems, using different instruction set extensions. In this section, we present the results of two problem instances, based on the implementations presented in [18]. The main objective was to analyze the impact of coding style and implementation strategy in a high-level programming language on the generated code, and the effectiveness of the implemented instruction set extensions. Although one needs to keep in mind that the chosen compiler framework and instruction set architecture both influence the relative performance of different configurations, some general trends could be identified.

The problem *find-first* asks to find the first array index, where the array element matches a given reference value. Implementation 1 and Implementation 2 are two standard implementations, which search

[2]The source code of this simulation environment is available on GitHub (`https://github.com/cgeyer/Sparc-V8-IS-extension-simulator`).

from the beginning to the end of the array, and stop as soon as a matching value is found. Implementation 3 resulted from the single-path transformation of the first version, and thus has a constant iteration count for the search loop. Implementation 5 realizes a different single-path algorithm that searches from the last element to the first, and thus also iterates over all array elements.

As can be seen in Figure 9, the execution time variability for the single-path versions is generally small, and zero if at least the conditional move instruction is available. Due to the stop condition in the loop body in Version 1 and Version 2, the adapted code generator of LLVM was not able to make use of any instruction set extensions. Therefore, the execution times are roughly the same on all targets, but vary considerably depending on the input vector. Version 3 performs much better if conditional moves are available, as this allows to eliminate branches in the loop body. Both Version 3 and Version 5 have a competitive worst-case performance, although it is still worse than that of the standard implementations in this example.

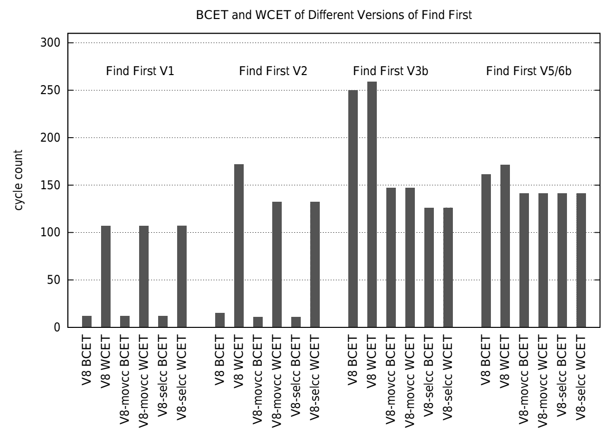

Fig. 9. Best- and worst-case execution times of the *find first* algorithm simulated on SPARC V8, SPARC V8 including conditional move and hardware loop instructions and SPARC V8 including conditional select and hardware loop instructions.

The second problem discussed here is *binary search* (Figure 10). Implementation 1 is a standard textbook implementation, and Implementation 8 is the result of applying the single-path transformation to it. Implementation 5 and Implementation 10 are two slightly different WCET-oriented algorithms, which are designed to iterate for a fixed number of times.

For this problem, the WCET-oriented algorithms benefit most from the instruction set extensions. The best solution with respect to worst-case performance is a single-path implementation with conditional select and hardware loop instructions: The execution time of the resulting machine code was independent of the input data, and improved by more than 40 % compared to the standard implementation without instruction set extensions.

The results of these experiments indicate that the coding style and the underlying instruction set has a significant impact on the execution time of the algorithms: Although all implementations solve the same problem, only a certain coding style allows the compiler to generate code with nearly constant execution time for all possible inputs. A constant execution time can only be achieved if the processor supports some kind of conditional move or select instructions. In some cases, the worst-case execution time of the single-path implementation is even better than the WCET of the original algorithm.

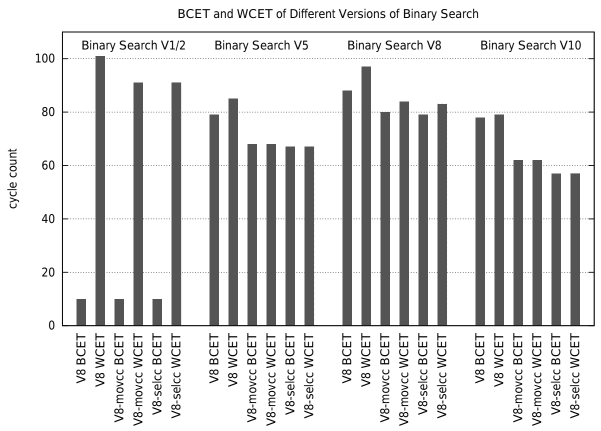

Fig. 10. Best- and worst-case execution times of the *binary search* algorithm simulated on SPARC V8, SPARC V8 including conditional move and hardware loop instructions and SPARC V8 including conditional select and hardware loop instructions.

B. Quantitative Evaluation of ISA Extensions

To assess the impact of the ISA extensions on the code size, the number of conditional branches and the worst-case cycle count quantitatively, a set of 24 algorithms has been considered. These algorithms were partly taken from the WCET benchmarks of the WCET group at the Mälardalen Real-Time Research Centre, a benchmark suite established in the WCET research community [19]. The other algorithms were developed by students of the Vienna University of Technology in the course of a WCET lecture. The algorithms were translated to assembly code for various *targets*, i.e., the SPARC V8 instruction set enhanced with different combinations of the instruction set extensions presented in Section IV. For each target and each algorithm, several key characteristics of the resulting assembly code have been evaluated, including the code size, the number of basic blocks, the number of conditional branches, and the worst- and best-case cycle count if applicable.

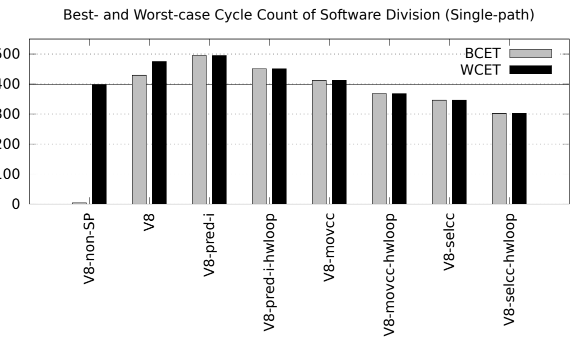

Fig. 11. BCET and WCET of several targets running a single-path implementation of a software-division algorithm. The first column *V8-non-SP* shows the performance on a target without any instruction set extension running a non-single-path implementation of the algorithm.

Figures 11 and 12 exemplify the impact of the various instruction set extensions on the approximated worst-case and best-case execution time on a software division algorithm, and a threshold algorithm,

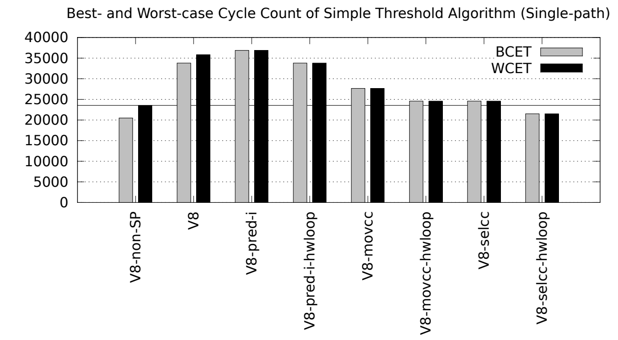

Fig. 12. BCET and WCET of several targets running a single-path implementation of a threshold algorithm. The first column *V8-non-SP* shows the performance on a target without any instruction set extension running a non-single-path implementation of the algorithm.

respectively. The first column shows the performance of a non-single-path variant, on a target without any extensions, constituting the baseline for the comparison. The targets with extensions were compared in following configurations: conditional move, select, and predicated blocks, each with and without hardware loops. For these two examples, which are representative for our benchmark suite, the compiler was able to produce code with low execution time jitter, with any of the instruction set extensions. The penalty resulting from the single-path transformation was compensated best by the target featuring conditional select instructions and hardware loop support, and was even outperforming the original solution in terms of WCET.

Predicated blocks are designed to be able to deal with nested input-data dependent conditional constructs. The performance penalty is inversely proportional to the size of the code they enclose. Code written in a WCET-oriented fashion ideally does not make use of input-data dependent branching. As most of the input-data dependent statements in the benchmark code were simple assignments, the high penalty of predicated blocks resulted in the low performance of targets featuring them. Consequently, the evaluation mainly focuses on the targets with only conditional move/select instructions and support for hardware loops. The detailed evaluation results are available in the appendix of [15].

Based on a trade-off between code size, number of conditional branches and worst-case cycle count, two targets outperforming all others in most of the named categories have been identified: A SPARC V8 with conditional move instructions and support for hardware loops and a SPARC V8 with conditional select instructions and support for hardware loops. The size of the generated code was low for both configurations – only the targets with conditional move or select instructions but without hardware loops produced more compact code in some cases.

Figure 13 shows the code size of the selected two targets relative to the original SPARC V8 target: In nearly all cases, the resulting code size is less or equal in comparison with the original solution. Moreover, single-path algorithms (indicated by the suffix *_sp*) tend to profit more of the instruction set extensions than the other ones. The main reason for the reduction in code size is the fact that the LLVM intermediate code representation features a select statement, which is used for many conditional assignments of the original C-code. For a target supporting a conditional select or a conditional move instruction, that intermediate level select statement is translated to the corresponding machine instruction. This is more compact than both branching code and predicated blocks, particularly for a single register copy machine instruction.

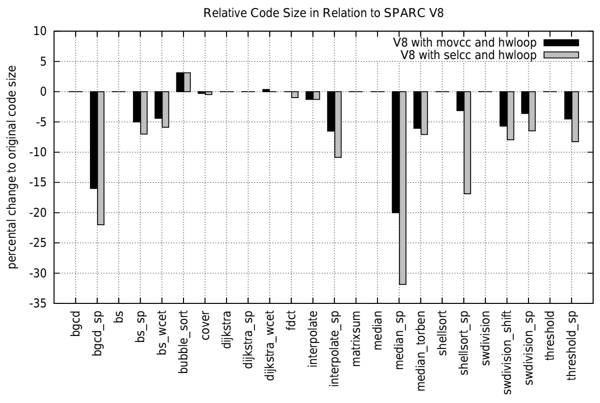

Fig. 13. Relative code size of 24 algorithms in relation to the original SPARC V8 instruction set without any extensions.

Figure 14 shows the number of conditional branches on the targets implementing conditional move and hardware loop instructions and of the target implementing conditional select and hardware loop instruction, in comparison to the assembly code on the SPARC V8 without any extensions. The more input-data dependent conditional branches a program contains, the more different control flow paths exist, each having a specific execution time. This increases the overall variability of the programs execution time, degrading its time predictability. For a majority of the evaluated algorithms, the number of conditional branches has been reduced. For the rest, the number remains equal. Like for the code size, the number of branches could be reduced due to the fact that LLVM internally makes use of a conditional select statement that can easily be translated to the conditional move or select instruction of the selected targets.

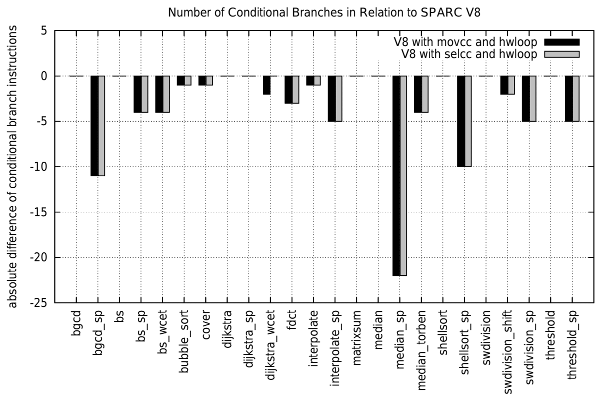

Fig. 14. Absolute number of conditional branches in comparison with the original SPARC V8 instruction set without any extensions.

Finally, we compare the worst-case performance of the targets with the instruction set extensions to the original SPARC V8. As illustrated in Figure 15, in 11 of the evaluated algorithms the worst-case performance has improved for targets implementing conditional move or select instructions in combination with hardware loops. Like for the code size and number of branches, the performance is better mainly on single-path algorithms. The reason for this is that the single-path implementations regularly make use of the ternary ?-operator in the C source code which can be translated directly to a conditional move or select instruction on targets supporting them. On the original SPARC V8 target, conditional branching code is used instead, which conflicts the goal to reduce input-data dependent branching, and hence hence the execution-time variability. Moreover, the worst-case path may take more cycles to execute than the single-path solution of the resulting assembly code of the two selected targets.

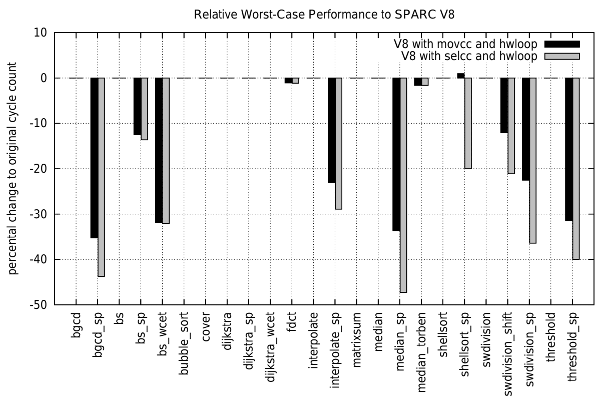

Fig. 15. Relative worst-case cycle count in comparison with the original SPARC V8 instruction set without any extensions.

VI. Conclusion

The presented evaluations show that the coding style in combination with the instruction set of the target architecture has a significant impact on the execution time of the compiled versions of the selected algorithms. Given different implementations of algorithms that solve the same problem, only a certain programming style allowed the compiler to generate code with almost constant execution time for all possible input vectors. A constant execution time, meaning the same number of execution cycles for all input-data scenarios, can only be achieved if the processor supports some kind of conditional move or select instructions. In some cases, the resulting worst-case execution time of a single-path variant of an algorithm can be better performing than the original algorithm.

For the investigated instruction-set extensions and their combinations, the evaluation of a number of algorithms and algorithm variants showed that the most promising candidates in terms of code-size, number of reduced branches and worst-case performance are targets that implement conditional move or select instructions in combination with hardware loops. These targets outperformed the other targets for most of the tested programs. This was due to the fact that WCET-oriented programs avoid using input-data dependent branching and ideally limit conditional execution to simple assignments, for which predicated blocks impose a higher penalty compared to conditional move and select instructions.

Acknowledgment

This research was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-Predictable Multi-Core Architecture for Embedded Systems (T-CREST).

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[2] P. Puschner and A. 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.

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

[4] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for Time Predictability," in *Proc. SAFECOMP 2012 Workshops (LNCS 7613).* Springer, 2012, pp. 382–391.

[5] P. Puschner, "Algorithms for dependable hard real-time systems," in *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.* IEEE Computer Society Press, 2003.

[6] J. Fauster, R. Kirner, and P. Puschner, "Intelligent Editor for Writing WCET-Oriented Programs," in *Proc. 3rd International Conference on Embedded Software*, 2003, pp. 190–205.

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

[8] W.-S. Gan and S. M. Kuo, *Embedded signal processing with the Micro Signal Architecture.* Hoboken, Canada: John Wiley & Sons, 2007, ch. 5 – Introduction to the Blackfin Processor, pp. 163 – 216.

[9] M. Schöberl, *JOP Reference Handbook.* CreateSpace, 2009.

[10] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a java processor," *Softw. Pract. Exper.*, vol. 40, no. 6, pp. 507–542, May 2010. [Online]. Available: http://dx.doi.org/10.1002/spe.v40:6

[11] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 264–265.

[12] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES '08. New York, NY, USA: ACM, 2008, pp. 137–146.

[13] N. Ip and S. Edwards, "A processor extension for cycle-accurate real-time software," in *Embedded and Ubiquitous Computing*, ser. Lecture Notes in Computer Science, E. Sha, S.-K. Han, C.-Z. Xu, M.-H. Kim, L. Yang, and B. Yiao, Eds. Springer Berlin / Heidelberg, 2006, vol. 4096, pp. 449–458.

[14] R. E. Gonzalez, "Xtensa: a configurable and extensible processor," *Micro, IEEE*, vol. 20, no. 2, pp. 60–70, March/April 2000.

[15] C. B. Geyer, "Instruction Set Extensions for Time-Predictable Code Execution," Master's thesis, Technische Universität Wien, 2012. [Online]. Available: http://media.obvsg.at/AC07813290-2001

[16] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. 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*, June 1995, pp. 138–149.

[17] J. C. Park and M. Schlansker, "On predicated execution," Hewlett Peckard Software and Systems Laboratory, Tech. Rep., May 1991.

[18] P. Puschner, "Evaluation of the Single-Path Approach and WCET-Oriented Programming," Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Research Report, January 2007.

[19] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks - Past, Present and Future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=2284