A Processor Extension for Time-Predictable Code Execution

Michael Platzer, Peter Puschner
Institute of Computer Engineering
TU Wien
Vienna, Austria
michael.platzer@tuwien.ac.at
peter@vmars.tuwien.ac.at

2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC) | 978-1-6654-0414-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISORC52013.2021.00016

***Abstract*—In this paper, we present an instruction filter, a simple architecture extension that adds support for fully predicated execution to existing processor cores that do not natively support it. This makes single-path code execution and hence high quality and easily derivable worst-case execution time (WCET) information available for a wide range of processors. We have implemented the single-path instruction filter for two processors and evaluated it on the TACLe benchmark collection. The results demonstrate that despite the seeming inefficiency of single-path code, our method does not substantially increase the WCET. Therefore, running single-path code on processors with our instruction filter represents a competitive method for time-predictable code execution.**

***Index Terms*—real-time systems, single-path code, predictable timing**

I. Introduction

In hard real-time systems, the execution of a task must complete within a time limit. Otherwise, the system might fail. Therefore, it is essential to guarantee that the task will not exceed that limit, which requires to bound its Worst-Case Execution Time (WCET). Bounding the execution time is usually done either through Static Timing Analysis (STA) or through measurement-based techniques. However, both of these approaches struggle with the large number of execution paths in typical programs [1].

In STA, the standard approach is to construct a graph in which nodes represent timing-relevant system states and edges represent the possible state transitions. This graph allows calculating the WCET via implicit path enumeration [2]. Every additional path in a program potentially increases the number of reachable states. Hence an increasing number of execution paths requires keeping track of a growing number of possible hardware states, thus leading to the state space explosion problem. The usual remedy for this are abstractions, such as value abstraction. While abstractions allow exploring larger system states, they reduce the precision of the analysis and lead to over-estimation of the actual WCET.

In measurement-based approaches, exhaustively measuring the duration of every path is infeasible in practice. Instead, statistical methods are used to infer a probabilistic WCET bound based on a timing model [3]. However, this bound is not safe since there is a non-zero probability that it might be violated, which is unacceptable for hard real-time systems.

Single-path code is a code generation paradigm that makes execution time bounding trivial by producing programs with a single execution trace [4]. Single-path code eliminates the need for abstractions that lead to over-approximations in static analysis and uncertainty in probabilistic approaches. The STA of a single-path program needs to keep track of one path only. On processors with data-independent instruction timings, the execution time of single-path code is constant with respect to input data. Hence a single measurement is sufficient to determine the WCET.

Single-path code uses predicated execution to enable or disable instructions conditionally, thereby replacing conditional control-flow instructions with predicated instructions [4]. Therefore, the target architecture must support predication. Most Instruction Set Architectures (ISAs) have some form of conditional instructions besides control-flow instructions, such as a *conditional move* instruction [5]. This allows a program to execute all branches in a conditional statement speculatively and then discard the results of all but one of the branches. However, speculative execution adds additional complexity to the code, particularly when a speculatively executed branch must avoid exceptions (e.g., division by zero). Therefore, to *efficiently* execute single-path code, the processor should support fully predicated execution, where all instructions can be enabled or disabled based on predicates.

Fully predicated execution is not a common feature in modern processor architectures [5]. The 32-bit ARM ISA is notable for supporting it by allowing every instruction to be enabled or disabled based on condition codes in the status register [6]. The limited availability of fully predicated execution confines single-path code to those few architectures. However, these might not always be the best fit for every application since other requirements could favor different execution platforms. In that case, one option is to build a custom processor with custom ISA, as has been done by Schoeberl et al. [7], who developed the Patmos processor within the T-CREST project. Patmos supports fully predicated execution, and the compiler backend written especially for it can produce single-path code that has an execution time that is effectively independent of input data. Developing a purpose-built processor with a dedicated instruction set is already a daunting and complex task in itself. On top of that, it requires

2375-5261/21/$31.00 ©2021 IEEE
DOI 10.1109/ISORC52013.2021.00016

building a custom toolchain capable of compiling programs to that new instruction set.

We would like to bring the benefits of single-path code to existing architectures without the need of building a new processor and developing a new instruction set. We propose a novel approach to extend existing processors with the ability to execute single-path code by adding an instruction filter in the instruction fetch path of the core. This requires minor modifications to the processor implementation that can easily be applied to a wide variety of architectures. As a result, these processors are able to execute temporally predictable code.

This paper makes the following contributions:

- We present a novel approach to add predicated execution to existing processor cores by adding an instruction filter with an internal predicate stack. The filter interprets the special instructions controlling the predicates at runtime and filters regular instructions fetched by the core depending on these predicates' state. Our instruction filter is implemented in SystemVerilog and will be made available under a liberal open-source license.
- We demonstrate the feasibility and evaluate our approach's performance by applying our single-path filter to two different processor cores: LEON3, a core using the SPARC v8 architecture [8], and the ARM Cortex-M0, which uses the 16-bit ARM Thumb ISA [6]. We synthesize the combined processing systems on a Field-Programmable Gate Array (FPGA) and compare the constant execution time of single-path versions of the TACLe benchmarks [9] with the WCET of the equivalent regular machine code versions.

This work is organized as follows: Section II gives a more detailed overview of the single-path paradigm, along with its advantages and drawbacks, and presents prior approaches to generating and executing single-path code. In Section III, we discuss the concept and requirements of an instruction filter for the execution of single-path code, and in Section IV, we explain details of our implementation. Section V presents our evaluation results based on the TACLe benchmark collection, and Section VI concludes this paper.

II. Background and Related Work

Although it is essential to determine the WCET of a task in critical real-time applications, actually determining a tight bound using STA remains a complex undertaking [10] [11] [12]. It requires solving two problems: modeling execution platform's timing behavior and determining the possible execution paths of a program [1]. While the severity of the first problem depends on the hardware's temporal predictability [13], the latter's complexity increases with the number of execution paths in the software.

Measurement-based methods were proposed as an alternative to STA [3]. These are usually hybrid approaches combining measurements with static analysis. For instance, in Measurement-Based Probabilistic Timing Analysis (MBPTA), timing measurements are used to build a hardware timing model complementing standard STA to determine WCET bounds [14]. While these approaches generally allow obtaining lower bounds, the accuracy of those bounds and their risk depends on hardware systemic effects [15] [16] and appropriate test coverage [17], i.e., the selection of execution paths of which the execution time was actually measured.

The number of possible program execution paths grows exponentially with the number of control-flow alternatives. Hence the analysis of all paths in STA and measuring the execution time of all paths quickly becomes intractable. Therefore, abstractions are needed which lead to over-estimation in STA and uncertainty in measurement-based approaches.

A. Single-Path Paradigm

Single-path code is a code generation paradigm in which all execution traces of a program are merged into a single execution path by eliminating all data-dependent control-flow changes [18], thus making timing analysis trivial. Single-path code executed on a processor with data-independent instruction timings has constant execution time regardless of input data. Therefore, task timing is repeatable [19], and the WCET can be determined with a single measurement or a trivial analysis step [4].

Instead of using control-flow instructions to execute code blocks conditionally, single-path code uses predicated execution to enable or disable individual instructions conditionally [4]. While the same sequence of instructions is executed by the processor every time a single-path program is run, the effect of instructions is controlled by the state of their predicates, which capture the truth values of conditions. Thus, predicated instructions replace the conditional control-flow instructions used in regular machine code.

Fig. 1 illustrates the concept of predicated execution. The pseudo-code on the left shows how in regular machine code, the control-flow for a simple conditional statement with two alternatives (i.e., an *if-then-else*-statement) is redirected based on the condition `COND` to execute either the first or the second assignment of the variable `x`. However, single-path code, shown on the right, uses predicated execution to enable one of the assignment instructions depending on the condition. Both branches of the conditional statement are executed, but the instruction disabled by a false predicate has no effects.

Function calls within conditional statements are also executed unconditionally, though the entire function is disabled if the associated condition evaluates to false. Therefore, recursive functions require recursion bounds to limit the maximum number of nested calls. Otherwise, a recursive function would

```
  goto else if ¬COND                    eval COND
  x = 1                          ( COND) x = 1
  goto end
else:
  x = 2                          (¬COND) x = 2
end:
  ...                                    ...
```

Fig. 1. A conditional statement as implemented in regular machine code using conditional control-flow instructions on the left and the equivalent single-path version which instead uses predicated execution on the right.

call itself over and over again indefinitely. Likewise, the WCET analysis of regular machine code needs the recursion bound of each recursive function to compute such a function's worst-case timing behavior.

In single-path code, loops are executed for a constant number of iterations. This constant number is the maximum iteration count of the loop, i.e., the loop bound. The loop condition predicates all instructions within a loop. Therefore, all subsequent iterations will have no effects once the loop condition becomes false, which corresponds to exiting the loop in regular code. Single-path code requires that the loop bound is known, as is required for timing analysis.

The drawback of single-path code is that all branches of conditional statements must be executed. Hence, the WCET of a conditional statement in single-path code is the sum of the individual branches, while for regular machine code, it is the maximum. Therefore, the execution time of single-path code is expected to be longer than that of regular code. However, if the WCET of the individual branches is short or there is only one branch with a large execution time, then the difference can become negligible. Single-path code always executes loops for the worst-case number of iterations. Hence the WCET of loops in single-path code remains the same as for regular code.

Algorithms used for STA typically over-estimate the WCET. This is due in large part to the abstractions required for analyzing large programs with lots of execution paths. Therefore, techniques reducing the number of execution paths, such as infeasible path elimination, can be used to refine WCET estimates [20]. Single-path code takes this approach to its extreme by reducing the program to one path only, with a constant execution time on time-predictable hardware. Thus the execution time (and hence WCET) of a single-path program can be lower than the WCET bound of the equivalent regular code, depending on the quality of the STA algorithm and timing model, as well as the structure of the program [21].

Apart from avoiding a complex WCET analysis, the use of single-path code in real-time systems has other advantages. For instance, determining a safe timing bound can be done on the hardware instead of on a timing model, thus eliminating mismatches between the model and the processor as a potential source for errors.

B. State of the Art in Single-Path Code Execution

Puschner et al. [22] first presented a method to transform regular code into single-path code (initially using the *conditional move* instruction) and subsequently formalized the approach and showed that every WCET-analyzable code can be converted to single-path code [23]. Schoeberl et al. [24] implemented the *conditional move* instruction on the time-predictable Java Optimized Processor (JOP), which has instruction timings that are free of data-dependencies. They demonstrated that single-path programs executed on this platform do indeed have a constant execution time.

Geyer et al. [25] investigated which ISAs and extensions thereof are suitable for the execution of single-path code both in terms of execution time and code size. They compared single-path code using partial predication with single-path code using full predication. For the latter, they implemented *predicated blocks* on the SPARC v8 architecture, a form of predication where an entire block of code is predicated. Although these predicated blocks could not be nested and thus only be applied to leaf statements, single-path code performance improved significantly compared with partially predicated execution.

While those early contributions focused on a description of the principles of single-path code, Prokesch et al. [26] analyzed single-path conversion on the Control-Flow Graph (CFG) level of a program. They introduced an algorithm to automatically generate single-path code for platforms that support fully predicated execution. Each basic block of the CFG is predicated according to the conditions that apply to it. Although Prokesch et al. implemented single-path transformation in the Patmos compiler [7], the algorithm itself works independently of the target architecture. We use this method to generate single-path code for various ISAs, which we extend with special instructions to control the state of the predicates.

Besides single-path code, other methods were proposed that implement cycle-accurate execution times, such as deadline instructions [27] suspending a program's execution until a certain time has elapsed. While this is a similarly lightweight processor extension as the one we propose, setting a deadline already requires one to know the WCET of a code section. This is also the case for other architectures that implement instructions for precise timing control, such as the PRET architecture [28], [29], and time-triggered programming models (e.g., the Giotto model [30]). By contrast, single-path code does not require a timing analysis of the code since its single execution trace naturally guarantees constant execution times with respect to input data on time-predictable hardware.

Table I compares methods for execution-time bounding in real-time systems. While STA provides safe execution time bounds, they are usually over-estimated. Probabilistic approaches allow for tighter bounds. However, these are not safe. Single-path code has constant execution time on platforms with data-independent instruction timings. Thus the WCET can be determined with a single measurement and does not require a platform-specific timing model. While earlier approaches implemented custom processor architectures for single-path support, our method outfits existing execution platforms with single-path support with minimal effort.

TABLE I
METHODS FOR EXECUTION-TIME BOUNDING ON TIME-PREDICTABLE ARCHITECTURES

Method	STA	Probab. Methods	Previous Single-Path	**Our work**
Safe WCET bounds	✓		✓	✓
Tight WCET bounds		✓	✓	✓
Applicable to existing processors	✓	✓		✓

III. Single-Path Filter

The goal of our work is to execute single-path code on existing processor cores. We want to take advantage of fully predicated execution to execute single-path code efficiently [5]. Since most ISAs do not support full predication, we extend those with special instructions that manipulate the state of predicates. These special instructions are encoded with unused opcodes of the respective instruction set.

We design an instruction filter that is added to the instruction fetch path of a processing core. This filter interprets the special predicate-defining instructions of the single-path code and filters regular instructions depending on the predicate states. Fetched instructions are either passed to the core or replaced by NOP instructions (there might be several instructions that have no effect, but we refer to all of them as NOPs for simplicity). As a result, the processor receives a stream of filtered native instructions (either instructions from the object code or NOPs). Fig. 2 shows a conceptual diagram of a processing platform using the filter. Instructions are only forwarded to the core if all predicates on the predicate stack are true. Otherwise, they are replaced by NOPs.

Single-path code requires the ability to conditionally modify predicates since these are used to capture the truth value of conditions. Therefore, the instruction filter needs access to the results of comparisons in the core. Most architectures use condition codes to capture the results of compares and evaluate conditions. Hence, we add an interface that routes the condition codes of the processing core into the single-path filter. This allows the filter to evaluate conditions analogously to the processor core and to modify the predicates accordingly.

The filter also manages a loop counter stack, which holds the iteration counters of loops in single-path code, a return address stack that stores the return addresses of single-path function calls and recursion counters (not shown in Fig. 2).

In our implementation, we require that all predicates on the predicate stack are true to enable instructions and thereby forward them to the core. Although our hardware implementation does not differentiate between different types of predicates, we distinguish them logically based on the purpose they serve in single-path code.

1) A *conditional predicate* is pushed to the stack for each conditional statement (e.g., *if-then-else* statements) and initialized based on the condition's result.
2) A *loop predicate* is used for each loop to capture the loop condition's truth value and any other conditional statements that exit the loop (e.g., *break*).
3) An *iteration predicate* is used alongside the loop predicate to disable the remainder of the loop body for one iteration only as a substitute for a *continue* statement.
4) Each function has a *function predicate* that disables the remainder of the function as a replacement to an early *return* statement in regular code.

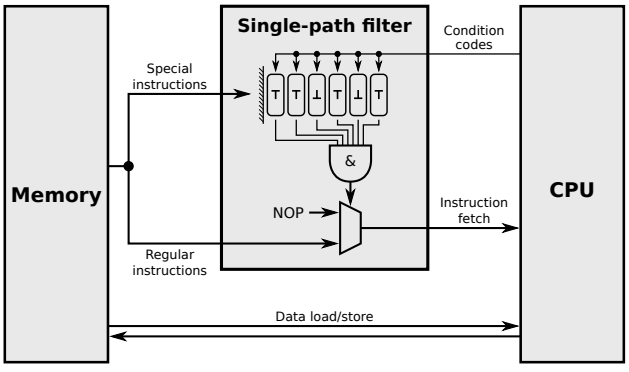


Fig. 2. Concept diagram of the single-path filter: Instructions are fetched from memory and pass through the filter, from where they are either passed on to the core or replaced by an instruction with no effects. Special instructions are used to control the predicates. The filter has access to the condition codes of the core, thus allowing to set predicates conditionally.

Fig. 3 shows the C code for a simple conditional statement, along with pseudo-assembler representations of its regular and its single-path variants. The generic operations *OP_A*, *OP_B*, *OP_C*, and *OP_D* represent instructions from the processor's native instruction set. In regular machine code, the conditional execution of either *OP_B* or *OP_C* is realized with control-flow instructions. In the single-path version, a conditional predicate is used instead, which is cleared if the condition *COND* is false (all predicates are initialized to true), thus capturing the truth value of *COND*. Hence, *OP_B* is only enabled if *COND* is true. The value of the predicate is subsequently inverted, thereby enabling *OP_C* only if *COND* is false. The right column shows the predicate stack state depending on *COND* for each of the generic operations.

Fig. 4 shows a similar representation for a simple loop. This time the single-path version also contains a control-flow instruction, a special instruction used in conjunction with a loop counter. The single-path filter substitutes this instruction either by a jump to the start of the loop as long as the loop counter is not 0 or by a NOP to exit the loop when the loop counter reaches 0. The loop counter is pushed to the loop counter stack and initialized with the loop bound specified in the annotation before the start of the loop and then decremented on every iteration. The loop predicate (at index 1 in the predicate stack) captures the truth value of the loop condition *COND_A* and the iteration predicate (at index 0) is cleared if *COND_B* evaluates to true, but is set again at the end of the loop body, thus substituting the *continue* statement. Note that a *break* statement ends the loop in the same fashion as if the loop condition would evaluate to false, hence in single-path code, it would simply clear the loop predicate.

Our single-path filter can conceptually be integrated into any processor core. We choose processors with data-independent instruction timings in order to have constant execution times for our single-path code. In particular, we do not use data caches to avoid varying latencies for memory accesses. Single-path code can be used on architectures with a data cache, however this requires a simplified timing analysis since the presence of a data cache might introduce input-dependent timing variability.

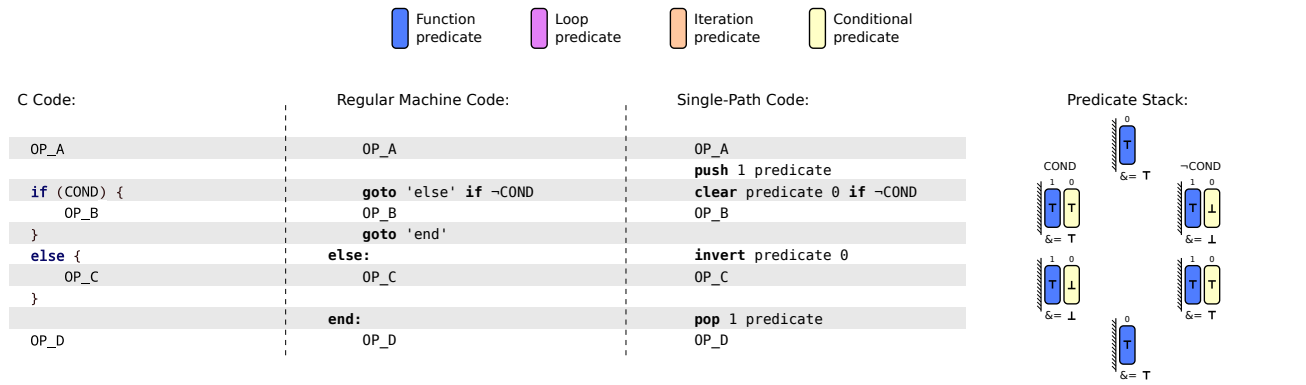


Fig. 3. Example of a conditional statement in single-path code: While regular machine code uses control-flow instructions to conditionally execute code, in single-path code predicates are used instead.

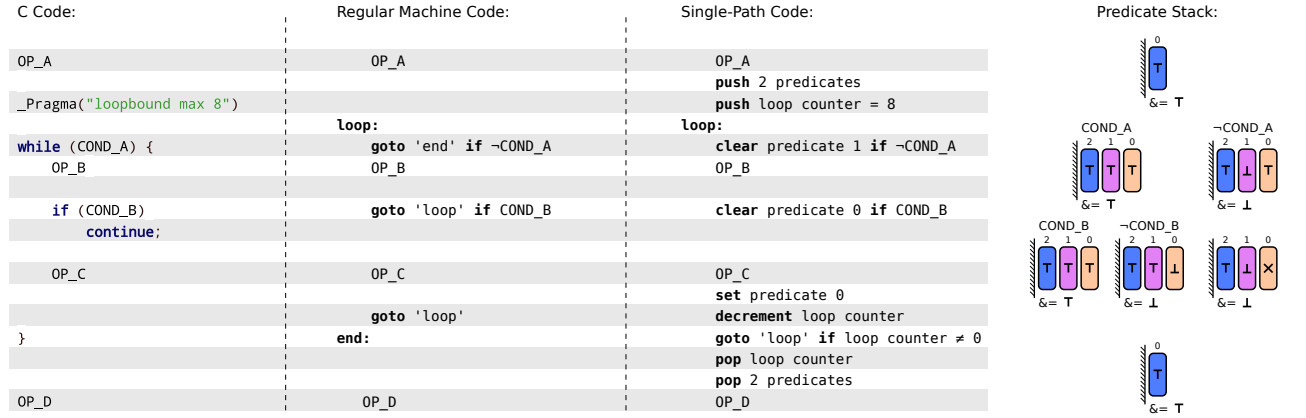


Fig. 4. Example of a loop in single-path code: The loop bound annotation is used to initialize the loop counter in single-path code and the loop is executed for a constant number of iterations. The loop predicate capturing the loop condition and the iteration predicate, which is cleared by a *continue* statement and reset at the start of each iteration, control whether the instructions are actually active.

IV. Implementation Details

We implemented our single-path filter in SystemVerilog and adapted it to extend the following two processors:

1) LEON3, an open-source SPARC v8 processor developed by Cobham Gaisler for safety-critical applications [8].
2) ARM Cortex-M0, a core developed by ARM, which uses the 16-bit ARM Thumb instruction set [6], available as an IP core via ARM's DesignStart program [31].

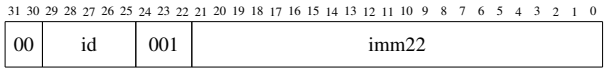
These two processors can be used as soft-cores on a FPGA and we synthesized a version with our single-path filter for both. This section details some implementation aspects, such as the special instructions used to control the filter and how disabled instructions are substituted with NOPs.

A. Extending ISAs with Special Instructions

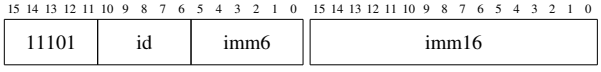
To control the single-path filter, the instruction set of a processor must be extended with special single-path instructions controlling the predicates as well as the loop counter and return address stacks within the filter. These special instructions are implemented individually for each ISA with unused opcodes reserved for custom extensions.

We implemented these special instructions for the SPARC v8 and the 16-bit ARMv6-M Thumb ISAs to support the LEON3 and the ARM Cortex-M0 processors, respectively. Fig. 5 shows the encoding for our special instructions.

The SPARC v8 ISA uses four major instruction formats, with bits 31 and 30 of each instruction word specifying this format. Format 2 (with bits 31 and 30 set to 00) has a 3-



(a) SPARC-v8 Single-Path Instructions Encoding



(b) ARMv6-M Thumb Single-Path Instructions Encoding (note that these are two 16-bit instruction words combined into a 32-bit word)

Fig. 5. Encoding formats for the special single-path instructions in the SPARC-v8 and the ARMv6-M Thumb instruction sets. The field *id* is used to identify the individual single-path instructions. For both architectures a total of 22 bits can be used to encode immediate values (see Table II for a list of single-path instructions and their respective use of the immediate field).

bit opcode field in bits 24 through 22, for which only four of the eight available opcodes are implemented. Opcode 0 is permanently reserved to indicate an invalid instruction, but opcodes 1, 3, and 5 are available for extensions. We use opcode 1 for our special instructions. Bits 29 through 25 select which special instruction to execute, and the remaining 22 bits are available for immediate values required by some instructions.

The ARMv6-M Thumb architecture uses 16-bit wide instruction words and some 32-bit instructions that span two instruction words. The first halfword of a 32-bit instruction has bits 15 through 13 set to 111, with the following two bits selecting the instruction class. We use the undefined instruction class 1. 27 bits remain to encode the special instruction specifics (id and immediate).

Table II lists the special instructions that we implemented to control the predicates and other stacks within the single-path filter. There are twelve special instructions in total, of which five are used to manipulate the predicate stack and predicate values, three instructions are used for loops in single-path code, two instructions for calling and returning from single-path code, and two instructions for keeping track of and limiting the recursion depth of recursive functions.

TABLE II
SPECIAL PREDICATE-DEFINING INSTRUCTIONS

Instruction	Description	Immediate Field
PRED PUSH	Push new predicates to predicate stack (initialized to true)	Number of predicates to push
PRED POP	Pop predicates from predicate stack	Number of predicates to pop
PRED SET	Set a predicate (change its value to true)	Index of the predicate to set
PRED INVERT	Invert a predicate (toggle its value)	Index of the predicate to invert
PRED CCLR	Conditionally clear a predicate (set its value to false)	3 bits: condition, rest: predicate index
LOOP PUSH	Push new loop counter to loop counter stack	Initial value of new loop counter
LOOP POP	Pop top loop counter from loop counter stack	Unused
LOOP BRNZ	Branch to start of loop if top loop counter is not zero, post-decrement top loop counter	Branch address (architecture-specific encoding)
SP CALL	Call a single-path function (push program counter to return address stack)	Call address (architecture-specific encoding)
SP RET	Return from single-path function (pop address from return address stack and jumps to that address)	Unused
RECUR ENTER	Enter a recursive function (increment the function's recurrence counter if it is below the recursion limit, otherwise return immediately)	Unique index for this recursive function
RECUR EXIT	Exit a recursive function (decrement the function's recurrence counter)	Unique index for this recursive function

B. Substituting Inactive Instructions with NOPs

Apart from filtering and interpreting the special instructions, the single-path filter has to disable regular instructions if any predicates are false. Disabled instructions and the special single-path instructions are replaced by NOPs. However, simply replacing each of these instructions with a generic NOP would lead to different execution times between the enabled and the disabled version, hence violating the single-path code's desired constant execution time. Therefore, it is essential to find substitute instructions with the same execution time as the enabled variant but having no effect (i.e., acting as a NOP).

1) Arithmetic Instructions: Several RISC instruction sets, such as the SPARC-v8 ISA, use a hard-wired zero register, i.e., a read-only register that holds the constant value 0. Writing to the zero register has no effect. Processors usually execute instructions with the zero register as destination register normally, but their result is discarded. This allows the single-path filter to disable an instruction by merely replacing its destination register with the zero register.

Architectures that have no zero register require other ways to disable arithmetic instructions while maintaining constant instruction timing. One approach is to find a NOP instruction with equal execution time. The ARMv6-M Thumb instruction set used by the ARM Cortex-M0 processor has no general-purpose zero register, but the execution time for all arithmetic instructions is one cycle. Hence, the single-path filter can simply substitute any disabled arithmetic instruction with the dedicated one-cycle NOP instruction of this ISA.

2) Memory Load and Store Instructions: Disabled load and store instructions should access the memory bus in the same way as if they were enabled to exert all latencies associated with the memory system, all while acting as a NOP.

For the SPARC v8 ISA we can handle memory loads similar to arithmetic instructions by substituting the target register with the zero register. However, we also need to guarantee that the load uses a valid memory address. For this purpose, we reserve a select memory location that we can address with the immediate field of load and store instructions alone. Fig. 6 shows the two instruction formats for memory access instructions and the substitutions (highlighted in bold) that the single-path filter performs when these instructions shall be disabled.

The situation is more complex for the ARM Cortex M0, which has no zero register. However, the ARM Thumb ISA defines stack-relative load and store instructions, allowing to address memory locations by adding a signed immediate to the stack pointer's current value. Therefore, instead of redirecting disabled loads and stores to a known fixed memory location, we instead redirect them to an address well below the stack pointer, to the memory region reserved for further stack growth. Collisions with actual data are unlikely unless a function allocates large amounts of data on the stack. A stack analysis can be carried out to verify that the redirected loads and stores do not interfere with any stack data.

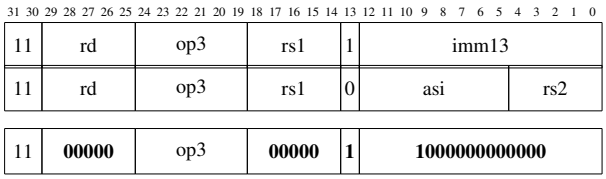


Fig. 6. Substituting disabled memory accesses for the SPARC v8 ISA: The two rows on the top show the two possible instruction formats for loads and stores, where the field *rd* identifies the target or source register respectively, *rs1* identifies the register holding the base address and either the immediate in *imm13* or the value in register *rs2* specifies a signed offset from the base address. The disabled version in the third row replaces the target/source register as well as the base address register with the zero register and encodes an offset of -4096 in the immediate field, thus redirecting the memory access to virtual address 0xFFFFF000. The opcode in field *op3* is not modified.

V. Evaluation

In this section, we evaluate our approach by comparing the single-path code's constant execution time with the WCET bounds of regular machine code. We added our single-path filter to the LEON3 [8] and the ARM Cortex-M0 [6] processors, synthesized the combined processing systems on a FPGA and measured the execution time of regular code and single-path on the FPGA using performance counters.

Both processors have a multi-stage in-order pipeline with predictable timings. We configured the LEON3 to use a single cycle load delay and disabled the data cache to guarantee data-independent instruction timings. In addition, data hazards are avoided statically by inserting NOPs. Hence, both processors execute a given single-path program in constant time.

The single-path code was generated following the method of Prokesch et al. [26], which was applied to the fully compiled and linked executable file generated by a target-dependent version of the GNU C Compiler (GCC) toolchain.

We evaluate the performance of our approach by comparing the constant execution time of single-path code with the WCET bound of the equivalent regular code on both processors for a set of benchmark programs. For regular machine code, the WCET is the relevant metric in real-time systems since this is the amount of processing time that system designers need to reserve for the execution of a task. For single-path code, however, the execution time is constant on hardware with data-independent instruction timing. Hence that constant execution time is the WCET of single-path code.

We used version 1.9 of the TACLe benchmark collection [9], a standard benchmark collection for WCET research, which comprises programs from several well-known benchmark suites, such as the Mälardalen [32] and the MiBench [33] collections. Every TACLe benchmark program contains annotations, such as loop bounds, which are required for WCET analysis and single-path code generation.

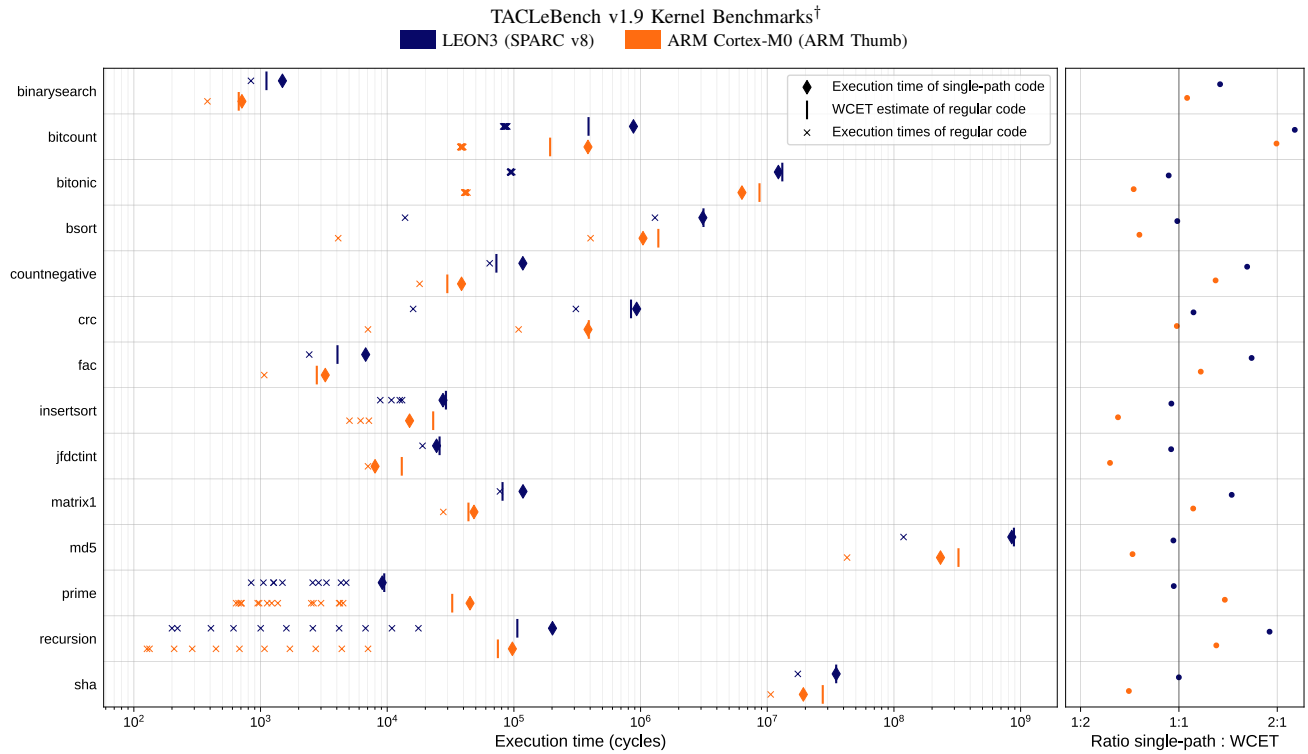
The WCET bounds were obtained using the *aiT WCET Analyzer* [34], a static timing analysis tool developed by AbsInt GmbH that computes tight WCET bounds [35]. Although the tool can extract some of the flow-facts (such as loop bounds) from the program executable through value analysis, annotations must be provided for all flow constraints that cannot be inferred automatically. The TACLe benchmark programs already contain all the required annotations. Thus, the WCET analysis uses exactly the same loop bounds and recursion limits that we also used for the single-path transformation.

Fig. 7 compares the constant execution time of the single-path version with the WCET bound of the regular version on the LEON3 and the ARM Cortex-M0 processors of each program of the kernel set of the TACLe benchmarks, except the programs using floating-point arithmetic (neither of the two processors has a floating-point unit). The large plot on the left shows the absolute execution time of the single-path versions and the WCET and samples of the execution time of the regular versions of each benchmark program for each processor on a logarithmic scale. The diamond symbol indicates the constant execution time of the single-path version, a vertical bar designates the WCET bound determined by the *aiT WCET Analyzer*, and crosses are used to show some measurements of the execution time of the regular version (note that for programs with constant input data the execution time of regular machine code is constant as well). A color code indicates the respective processor. The narrow plot on the right shows the ratio between the single-path code's execution time and the WCET of the regular machine code. A ratio of 1:1 indicates that the constant execution time of the single-path version is equal to the WCET bound of the regular version. The ratio is larger if the single-path code's runtime is larger than the WCET of the regular code and smaller if the single-path version is faster than the WCET bound.

The results show that the single-path version's execution time is sometimes higher and sometimes lower than the WCET bound of the regular version. For most of the benchmark programs, the relative difference is within a factor of two. Whether or not single-path code is faster depends on the program itself (for some programs, such as *bitonic*, the single-path code version is faster on both processors, while for others, e.g., *bitcount*, the single-path version is slower on both processors) and on the processing platform, since on the LEON3 processor the ratio between the runtime of the single-path version and the WCET of the regular version is almost always slightly higher than for the ARM Cortex-M0.

A program's structure, particularly the number and length of conditional statements, strongly influences the single-path code's execution time. As mentioned in Section II, a conditional statement's execution time in single-path code is the sum of all branches' execution times. In regular code, however, the WCET of a conditional statement is the maximum of the individual branches' execution times. Therefore, single-path versions of programs having conditional statements with multiple lengthy branches tend to have higher execution times than the WCET of regular machine code.

We further observe that single-path code performs better (when comparing it to the WCET of regular code) on the ARM Cortex-M0 than on LEON3. We assume that the structure of the machine code produced by the compiler's architecture-specific variants is similar enough between the two architec-



† Excluding programs using floating-point arithmetic.

Fig. 7. Comparison of the execution time of the single-path version with the WCET bound of the regular version of the TACLeBench benchmark programs. The large plot on the left shows the constant execution time of the single-path version of each benchmark program as well as the WCET bound and execution time of the regular code version of that program in CPU cycles. The narrow plot on the right shows the ratio between the execution time of the single-path version and the WCET of the regular version for each program. The WCET bounds have been obtained with the *aiT WCET Analyzer* from AbsInt GmbH.

tures that it does not fully explain the observed variations. It rather seems that the accuracy of the WCET bounds calculated by the timing analysis software varies between processors, possibly because some architectures are easier to analyze than others.

As explained in Section II, single-path code cannot reduce the WCET of a program, except for marginal improvements if the overhead of manipulating predicates is less than the overhead created by conditional control-flow instructions in the regular machine code version of a program. However, STA does not determine the exact WCET of a program but instead calculates a safe bound thereof. Hence, when the execution time of single-path code is lower than the WCET bound of the equivalent regular code, this is primarily due to an overestimation of the effective WCET by the analysis software.

While we did not apply any changes to the benchmark programs, it is likely that by optimizing the programs for single-path execution (for instance, by avoiding code duplication in the branches of conditional statements), the execution time of single-path code relative to the WCET of regular code can be further improved.

VI. Conclusion

We have presented a method to enable existing processor cores to efficiently execute single-path code by placing a filter in the instruction-fetch path of the core, which provides predicated execution by filtering instructions depending on the values of predicates. This allows real-time tasks to be executed on a wide variety of architectures since the highly predictable nature of single-path code means that their WCET can be determined with a single measurement, altogether avoiding the need for static analysis.

We have implemented the single-path generation and the instruction filter for two Reduced Instruction Set Computer (RISC) processors and evaluated the performance on the TACLe benchmark collection. Our comparison with the WCET bound of regular machine code shows that single-path code executes between twice as slow and twice as fast for most of the benchmark programs, depending on the structure of a program and the quality of its WCET bounds. Despite the perceived inefficiency of unconditionally executing all branches, the constant execution-time of single-path code is comparable to the WCET bound of regular machine code. Further performance gains are likely possible through targeted optimizations of the program code.

Our single-path filter enables existing processor cores to execute single-path code efficiently, thus allowing us to deploy single-path code on processors for which no WCET tool is available, with the potential to run code with precise timing guarantees on these processors.

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, no. 3, May 2008. doi: 10.1145/1347375.1347389

[2] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, 1997. doi: 10.1109/43.664229

[3] L. Santinelli, F. Guet, and J. Morio, "Revising measurement-based probabilistic timing analysis," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2017, pp. 199–208. doi: 10.1109/RTAS.2017.16

[4] P. Puschner, "The single-path approach towards WCET-analysable software," in *IEEE International Conference on Industrial Technology, 2003*, vol. 2, Dec. 2003, pp. 699–704 Vol.2. doi: 10.1109/ICIT.2003.1290740

[5] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, p. 138–150. doi: 10.1145/223982.225965

[6] D. Jagger, *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, 1996.

[7] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015. doi: 10.1016/j.sysarc.2015.04.002

[8] J. Andersson, J. Gaisler, and R. Weigand, "Next generation multipurpose microprocessor," 2010.

[9] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[10] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz, "Static WCET analysis of real-time task-oriented code in vehicle control systems," in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, Nov 2006, pp. 212–219. doi: 10.1109/ISoLA.2006.63

[11] A. Ermedahl, J. Gustafsson, and B. Lisper, "Experiences from industrial WCET analysis case studies," in *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, ser. OpenAccess Series in Informatics (OASIcs), R. Wilhelm, Ed., vol. 1. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007. doi: 10.4230/OASIcs.WCET.2005.811

[12] J. Gustafsson and A. Ermedahl, "Experiences from applying wcet analysis in industrial settings," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, 2007, pp. 382–392. doi: 10.1109/ISORC.2007.36

[13] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, July 2009. doi: 10.1109/TCAD.2009.2013287

[14] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," *Communications in Computer and Information Science*, vol. 17, pp. 430–444, 10 2008. doi: 10.1007/978-3-540-88479-8_30

[15] F. Guet, L. Santinelli, and J. Morio, "On the reliability of the probabilistic worst-case execution time estimates," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016.

[16] S. Milutinovic, J. Abella, and F. J. Cazorla, "On the assessment of probabilistic WCET estimates reliability for arbitrary programs," *EURASIP Journal on Embedded Systems*, 2017. doi: 10.1186/s13639-017-0076-8

[17] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 189–199. doi: 10.1109/ECRTS.2016.21

[18] P. Puschner and A. Burns, "Writing temporally predictable code," in *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*, Jan. 2002, pp. 85–91. doi: 10.1109/WORDS.2002.1000040

[19] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl, "A disruptive computer design idea: Architectures with repeatable timing," in *2009 IEEE International Conference on Computer Design*, 2009, pp. 54–59. doi: 10.1109/ICCD.2009.5413177

[20] B. Blackham, M. Liffiton, and G. Heiser, "Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014, pp. 169–178. doi: 10.1109/RTAS.2014.6926000

[21] P. Puschner, "Experiments with wcet-oriented programming and the single-path architecture," in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2005, pp. 205–210. doi: 10.1109/WORDS.2005.36

[22] P. Puschner, "Transforming execution-time boundable code into temporally predictable code," in *Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems*, 2002, p. 163–172. doi: 10.1007/978-0-387-35599-3_17

[23] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for time predictability," in *Computer Safety, Reliability, and Security*, F. Ortmeier and P. Daniel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 382–391.

[24] M. Schoeberl, P. Puschner, and R. Kirner, "A single-path chip-multiprocessor system," in *Software Technologies for Embedded and Ubiquitous Systems*, S. Lee and P. Narasimhan, Eds., 2009, pp. 47–57. doi: 10.1007/978-3-642-10265-3_5

[25] C. B. Geyer, B. Huber, D. Prokesch, and P. Puschner, "Time-predictable code execution — instruction-set support for the single-path approach," in *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, 2013, pp. 1–8. doi: 10.1109/ISORC.2013.6913195

[26] D. Prokesch, S. Hepp, and P. Puschner, "A generator for time-predictable code," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, April 2015, pp. 27–34. doi: 10.1109/ISORC.2015.40

[27] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," in *Embedded and Ubiquitous Computing*, 2006, pp. 449–458. doi: 10.1007/11802167_46

[28] 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*, 2008, p. 137–146. doi: 10.1145/1450095.1450117

[29] E. Lee, J. Reineke, and M. Zimmer, "Abstract PRET machines," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 1–11. doi: 10.1109/RTSS.2017.00041

[30] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003. doi: 10.1109/JPROC.2002.805825

[31] "DesignStart – ARM," https://www.arm.com/resources/designstart, accessed: 2021-04-16.

[32] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, vol. 15, 01 2010, pp. 136–146. doi: 10.4230/OASIcs.WCET.2010.136

[33] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. USA: IEEE Computer Society, 2001, p. 3–14.

[34] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Building the Information Society*, R. Jacquart, Ed. Boston, MA: Springer US, 2004, pp. 377–383.

[35] J. Gustafsson, "The worst case execution time tool challenge 2006," in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, 2006, pp. 233–240.