An Instruction Filter for Time-Predictable Code Execution on Standard Processors

Michael Platzer[0000−0002−5103−8848] ✉ and Peter Puschner[0000−0002−2495−0778]

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

Abstract. Dependable cyber-physical systems usually have stringent requirements on their response time, since failure to react to changes in the system state in a timely manner might lead to catastrophic consequences. It is therefore necessary to determine reliable bounds on the execution time of tasks. However, timing analysis, whether done statically using a timing model or based on measurements, struggles with the large number of possible execution paths in typical applications. The single-path code generation paradigm makes timing analysis trivial by producing programs with a single execution path. Single-path code uses predicated execution, where individual instructions are enabled or disabled based on predicates, instead of conditional control-flow branches. Most processing architectures support a limited number of predicated instructions, such as for instance a conditional move, but single-path code benefits from fully predicated execution, where every instruction is predicated. However, few architectures support full predication, thus limiting the choice of processing platforms. We present a novel approach that adds support for fully predicated execution to existing processor cores which do not natively provide it. Single-path code is generated by restructuring regular machine code and replacing conditional control-flow branches with special instructions that control the predication of subsequent code. At runtime an instruction filter interprets these predicate-defining instructions, computes and saves predicates and filters regular instructions based on the predicate state, replacing inactive instructions with a substitute that has no effect (e.g. a NOP). We are implementing this single-path filter for the LEON3 and the IBEX processors.

Keywords: Single-Path · Real-Time · Predictable Timing.

1 Introduction

In many dependable cyber-physical systems the execution of a task must complete within a time limit, otherwise the system might fail. It is therefore essential to guarantee that the task will not exceed that limit, which requires to bound its Worst-Case Execution Time (WCET). This 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 [14]. In STA every additional path requires to keep track of an ever growing number of possible hardware states thus leading to the state space explosion problem. In measurement-based approaches, exhaustively measuring the duration of every path is infeasible in practice.

Single-path code is a code generation paradigm which makes execution time bounding trivial by producing a program with a single execution trace [6]. The STA of a single-path program needs to keep track of one path only. On time-predictable hardware the execution time of single-path code is constant, hence a single measurement is enough to determine the execution time. Single-path code makes use of predicated execution to enable or disable instructions conditionally and thereby replace conditional control-flow branches with predicated instructions [1].

Since single-path code relies on predicated execution for conditional parts of a program, the target architecture must support that. Most Instruction Set Architectures (ISAs) have some form of conditional instructions besides control-flow instructions, such as for instance a *conditional move* instruction [4]. That allows to execute all traces in a conditional statement speculatively and then discard the results of all but one trace. However, this adds additional complexity to the code, in particular when a speculatively executed trace must avoid exceptions (e.g. division by zero). Therefore, in order 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 [4]. The ARM ISA is notable for supporting it, by allowing every instruction to be enabled or disabled based on condition codes in the status register. 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. [11], who developed the Patmos processor. Patmos supports fully predicated execution and the compiler backend written especially for it has the option to 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 on itself. On top of that it requires to build a custom toolchain which can compile programs to that new instruction set.

We would like to bring the benefits of single-path code to existing architectures without the need to build a new processor and to develop a new instruction set. We also want to avoid tying single-path code generation to a specific compiler. Therefore, we apply the single-path transformation as a post-processing step to the fully compiled and linked executable file of a program. We propose a novel approach to upgrade existing processors with the ability to execute single-path code generated that way by adding an instruction filter in the instruction fetch path of the processor core. This requires minor modifications to the hardware that can easily be applied to a wide variety of processor architectures.

To demonstrate the feasibility of our approach we have started to apply these changes to two processors: LEON3, a core using the SPARC v8 ISA and IBEX, which uses the RISC-V architecture.

This paper makes following contributions:

- We adapt the single-path generation algorithm of the Patmos compiler such that it can be used to convert machine code of various ISA to single-path code. Special instructions controlling the state of predicates are encoded with unused opcodes.
- 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 instructions fetched by the core depending on the state of these predicates.

This work is organized as follows: Section 2 gives a more detailed overview of the single-path paradigm, along with its advantages and drawbacks. Section 3 presents prior approaches to generating and executing single-path code. In section 4 we discuss the concept and requirements of an instruction filter for the execution of single-path code and in section 5 we explain details of our implementation. Section 6 describes the current state of our implementation and section 7 concludes this paper.

2 Single-Path Paradigm

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. It requires solving two problems: modelling the timing behavior of the execution platform and determining the possible execution paths of a program [14]. While the severity of the first problem depends on the temporal predictability of the hardware, the complexity of the latter increases with the number of execution paths in the software.

Measurement-based methods were proposed as an alternative to STA [10]. These are usually hybrid approaches, combining measurements with static analysis. For instance, Measurement-Based Probabilistic Timing Analysis (MBPTA) has been introduced by Wenzel et al. [13], where timing measurements are used to build a hardware timing model which complements standard STA for determining WCET bounds. While these approaches generally allow to obtain lower bounds, the accuracy of those bounds and their respective violation probabilities depends on hardware systemic effects and appropriate test coverage [3], i.e. the proportion 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 analysis of all paths in STA as well as measuring the execution time of all paths quickly becomes intractable. Single-path code is a code generation paradigm in which all execution traces

of a program are merged into a single execution path [6], thus making timing analysis trivial. Single-path code executed on a time-predictable processor has constant execution time regardless of input data and therefore the WCET can be determined with a single measurement.

Instead of conditionally executing code blocks using branches, single-path code makes use of predicated execution to conditionally enable or disable individual instructions [1]. While the same sequence of instructions is executed by the processors every time a single-path program is run, instructions might have no effects depending on the state of predicates. These predicates capture the truth values of conditions and thus predicated instructions replace the conditional branches used in regular machine code. Loops are executed for a constant number of iterations based on a loop bound. It has been shown that every WCET-analyzable code can be converted to single-path code [9].

The drawback of single-path code is that all execution traces must be executed, which is why the execution time of single-path code is typically larger than that of regular code. However, reduced performance is traded for increased predictability. Also, the increased execution time is often lower than the WCET bound of the equivalent regular code [7].

3 Related Work

A method to transform regular code into single-path code for platforms which support partial predicated execution has first been described by Puschner et al. [8]. It makes use of the *conditional move* instruction, which is implemented in several processor architectures. Conditional code sections are always executed speculatively regardless of the truth value of the respective condition, but the results are discarded if that condition is false.

In order to achieve constant execution times with respect to input data, single-path code must be executed on time-predictable hardware. Schoeberl et al. [12] implemented the *conditional move* instruction on the time-predictable Java Optimized Processor (JOP). They demonstrated that single-path programs executed on this platform do indeed have a constant execution time regardless of input values.

Geyer et al. [2] investigated which ISA and extensions thereof are particularly suitable for the execution of single-path code both in terms of execution time and code size. In particular, they compared single-path code using partial predication, which made use of a conditional move instruction as did earlier work, with single-path code using full predication. For the latter they implemented *predicated blocks* for the SPARC v8 architecture, a form of predication where an entire block of code is predicated by specifying a condition that would apply to subsequent code with a special *predbegin* instruction. The predication remains active until an associated *predend* instruction is encountered.

While those early contributions focused on a description of the principles of single-path code, Prokesch et al. [5] analyzed single-path conversion on the Control-Flow Graph (CFG) level of a program and 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. All instructions from the original regular machine code are kept, with the exception of conditional branches, which are replaced by special instructions that modify the predicates. The algorithm also retains the topological ordering of instructions, hence all active instructions are executed in the exact same order in which they would have been executed in the equivalent regular code. The algorithm was embedded into their port of the LLVM compiler which produces code for the time-predictable processor Patmos.

Although Prokesch et al. implemented single-path transformation in the Patmos compiler, the algorithm itself works independent of the target architecture. Since it operates on the CFG of a program and does not require access to the source code, it can also be applied as a post-processing step to the executable file produced by an arbitrary compilation toolchain. We use this method to generate single-path code for various ISA, which we extend with special instructions to control the state of predicates.

4 Single-Path Extension

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 [4]. Since most ISA 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 ISA.

We use the automated single-path transformation algorithm developed by Prokesch et al. [5] to convert regular machine code to single-path code. While Prokesch et al. implemented single-path generation inside their port of the LLVM compiler, we apply this transformation as a post-processing step to a fully compiled and linked executable. That way the single-path conversion is not tied to a specific compilation toolchain. The transformation rearranges the basic blocks of the CFG of a program and replaces conditional control-flow branches with special instructions that modify predicates.

Existing cores do not understand these special instructions, thus requiring some modifications. In an attempt to keep the required changes to a minimum, we design an instruction filter which interprets the special instructions controlling the predicates and implements predication for all other instructions by filtering out the inactive ones based on predicate values identified by the filter. It either passes fetched instructions to the processor or replaces them with NOP instructions (depending on the architectures there might be several instructions that have no effect, but for simplicity we refer to all of them as NOPs).

Allowing existing processor cores to execute single-path code therefore involves two steps:

1. Single-path code is generated from regular machine code by applying the method of Prokesch et al. to the executable file of a program. This single-path

code consists of restructured object code that includes special instructions for computing predicates.
2. An instruction filter is added to the processor core. At runtime, this filter interprets the special instructions of the single-path code, to compute predicates and filter instructions depending on the actual predicate states. As a result the processor receives a stream of filtered native instructions (either instructions from the object code or NOPs) at runtime.

The filter is placed on the instruction fetch path, such that all instructions pass through it as they are fetched by the core. Figure 1 shows a concept 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. Conditionally modifying predicates requires access to the condition codes of the processor, therefore we add an interface that routes the condition codes out of the core and into the filter. The remainder of this section discusses the requirements for such a single-path filter.

We apply the single-path conversion to the executable file of a program, after all compilation and linking steps have been completed, which has the advantage that it is not dependent on a specific compilation toolchain. That requires, however, that any additional state information necessary for the execution of the single-path code (such as for instance the predicate values) need to be saved in the filter, as saving it in memory or registers might lead to collisions with the memory or register allocation of the preceding compilation or linking steps.

The first requirement to execute single-path code generated in this manner is that the execution platform must support fully predicated execution. The single-path filter must interpret special instructions that compute predicates, manage

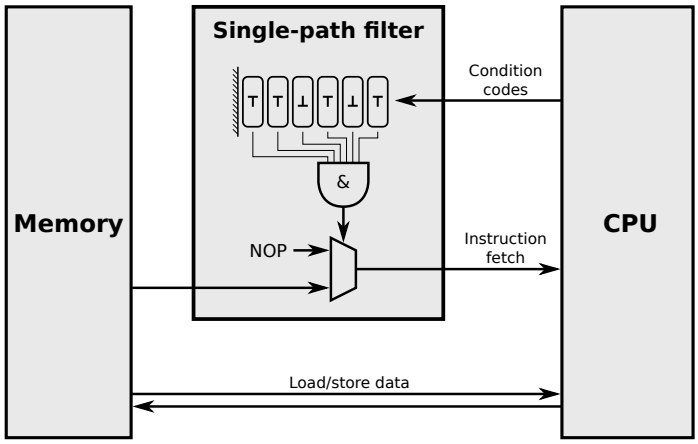

Fig. 1. Concept diagram of the single-path filter: Instructions are fetched from memory and pass through the filter, from where they are either passed on to the core or replaced by an instruction with no effects. The filter has access to the condition codes of the core, thus allowing to set predicates conditionally.

the predicates and filter out instructions that are disabled by these predicates. Predicates capture the truth values of conditions and a new predicate is required for every condition that we encounter. Predicates expire when the execution of subsequent instructions no longer depends on the associated condition. Programming constructs that use conditions, such as conditional statements (e.g. *if-then-else* statements) or loops, can be nested, with new conditions applying on top of others. Consequently the predicates should be managed in a predicate stack. A new predicate is pushed to the stack when encountering a condition and the predicate is removed from the stack when it expires. That predicate stack must be stored in dedicated hardware in the filter, such that the predicate values are readily available to it.

Apart from fully predicated execution, another requirement is that loops require an iteration counter. In regular code the number of iterations of a loop depend on the loop condition only. However, in single-path code the loop bound dictates the number of iterations and a counter is required to count these iterations. This counter cannot be stored in memory or a register either, since we do not want to restrict the hardware resources available to the compiler. Therefore, the iteration counters for loops in single-path code also need to be stored in hardware. Loops might be nested, hence instead of a single loop counter a loop counter stack is required. A new loop counter is pushed to that stack upon entering a loop and initialized with the total iteration count. The counter is then decremented on every iteration. When the loop counter reaches 0 the loop exits and the loop counter is removed from the stack.

Finally, the single-path filter must have a dedicated return address stack for single-path functions. In regular code a function call writes the address of the call instruction to a specific register known as the *return address register*. When the function returns, it transfers control back to that address. Function calls can be conditional, for instance when they appear inside conditional statements. In single-path code, every function call is executed unconditionally, but depending on the values of predicates, all instructions of that function might be inactive and thus the function call might have no effects. This is equivalent to a function that would not have been executed in regular code. Since an inactive function does not modify any memory locations or registers, including the return address register, the return address would be lost if it were not saved elsewhere. Therefore, the return address of single-path function calls must be stored in the filter as well. Function calls are usually nested, hence a return address stack is required.

5 Filter Implementation

In order to add the ability to execute single-path code to an existing processor, we add an instruction filter with a predicate stack which computes and saves predicates triggered by special predicate-defining instructions and filters regular instructions based on the values of these predicates, by either passing them on to the core or replacing them by instructions that have no effects (NOPs). The filter also manages a loop counter stack which holds the iteration counters of loops in

single-path code and a return address stack which stores the return addresses of single-path function calls. To control the behavior of the single-path filter the instruction set must be extended with special single-path instructions, which modify the state of these hardware stacks. Unused opcodes in the instruction set are used to encode these special instructions, which replace conditional control-flow branches when generating single-path code and are parsed and applied directly by the filter when fetched by the processor core.

Single-path code requires the ability to conditionally modify predicates, since the predicates are used to capture the truth value of conditions. Therefore, the instruction filter needs access to the results of comparisons in the core. On most architectures condition codes are used to capture the results of compares and to evaluate conditions. Hence, by giving the instruction filter access to these condition codes, it can evaluate conditions based on these condition codes analogously to the processor core and modify predicates accordingly.

Individual predicates are pushed to the predicate stack, where they are then modified either conditionally or unconditionally, thereby enabling and disabling the execution of subsequent instructions. Predicates are removed from the predicate stack in the reverse order than they have been added to it.

In our implementation we require that all instructions on the predicate stack are true in order 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. *Function predicates:* Each function has a function predicate, which is the first predicate pushed to the predicate stack upon entering a function and conversely the last predicate popped from the stack upon leaving that function. The function predicate is initially true and changes to false when the code encounters a *return* statement. Single-path code requires that all instructions of a function are always executed, hence an early return from a function is realized by clearing the function predicate and thereby causes the filter to substitute all remaining instructions of that function by NOPs.
2. *Conditional predicates:* A conditional predicate is pushed to the stack for each conditional statement (e.g. *if-then-else* statements). The conditional predicate is initialized based on the result of a condition and remains on the stack for as long as the condition applies.
3. *Loop predicates:* Each loop has a loop predicate. Similar to a function predicate, this is the first predicate pushed to the predicate stack when entering a loop and the last predicate removed when exiting the loop. The loop predicate is true as long as the loop condition is true. Once cleared it remains false for all remaining loop iterations.
4. *Iteration predicates:* In addition to the loop predicate, every loop also has an iteration predicate. The iteration predicate is set to true at the beginning of each loop iteration and is cleared if one iteration of the loop is aborted without exiting the loop, such as would happen when encountering a *continue* statement.

The following examples illustrate the use of predicates for conditional statements and for loops.

Figure 2 shows the C code for a simple conditional statement, along with pseudo-assembler representations of the regular version as well as of the single-path version of the machine code for that conditional. The generic operation *OP_A* is executed unconditionally prior to the conditional block. *OP_B* is executed if the condition *COND* is true, otherwise *OP_C* is executed instead. Finally, *OP_D* comes after the conditional block and is again executed unconditionally. In regular machine code the conditional execution of either *OP_B* or *OP_C* is realized with control-flow instructions. A conditional branch moves control to the *else* label if *COND* is false, thus executing *OP_C*. Otherwise *OP_B* is executed and then an unconditional jump brings control to the end of the conditional block. The single-path version, by contrast, does not use any control-flow instructions. Instead a new predicate is pushed to the stack and that predicate (with index 0 since it is at the top of the stack) is cleared if *COND* is false. Hence, the predicate at the top of the stack initially corresponds to the truth value of *COND*, and therefore the operation *OP_B* is only enabled if *COND* is true. Then, the value of the predicate is inverted, thereby enabling *OP_C* if *COND* is false. The right column shows the state of the predicate stack depending on the truth value of *COND* for each of the generic operations.

Figure 3 shows a similar representation for a simple loop. This time however the single-path version also contains a control-flow instruction. This is a special instruction that is used in conjunction with a loop counter, which will be replaced either by an unconditional jump to the beginning 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. Simultaneously, the loop counter is decremented by one every time control jumps back to the start of the loop. 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. Loops use a loop predicate to capture the state of the loop condition and an iteration predicate that is replaces branches to the start of the loop. While the loop predicate at index 1 in the predicate stack is cleared once if the loop condition *COND_A* is false and then remains false for all remaining iterations, the iteration predicate at index 0 is conditionally cleared if *COND_B* is true for one loop iteration only and is reset to true for the next iteration. Both of these predicates are pushed to the predicate stack before the beginning of the loop and removed from the stack after the loop has been left.

The single-path filter substitutes the instructions fetched from memory by instructions that have no effects (NOPs) when any of the predicates on the stack is false. In order to achieve constant execution time that substitute instruction must also have the same execution time than the original instruction. Which and how many instructions are used for this purpose will therefore depend on the specific processor. On architectures that use a hard-wired zero-register (i.e. a register that always reads as 0 and cannot be written) the destination register of an instruction can simply be replaced by that zero register, in which case the instruction does not modify any registers and thus has no effect.

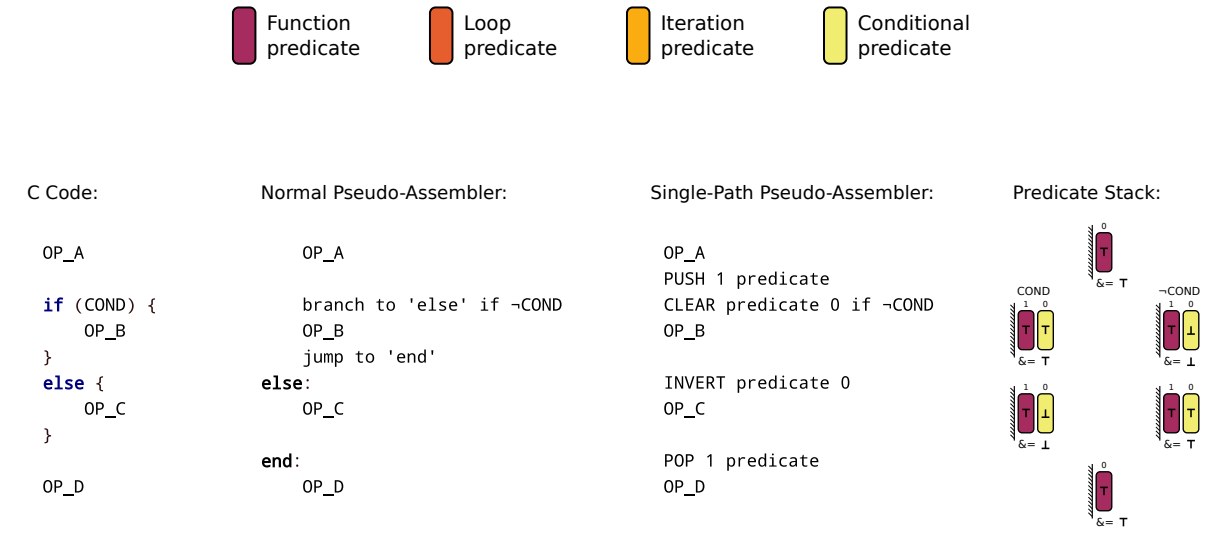

Fig. 2. Example of a conditional statement in single-path code: While regular machine code uses control-flow branches to conditionally execute code, in single-path code predicates are used instead.

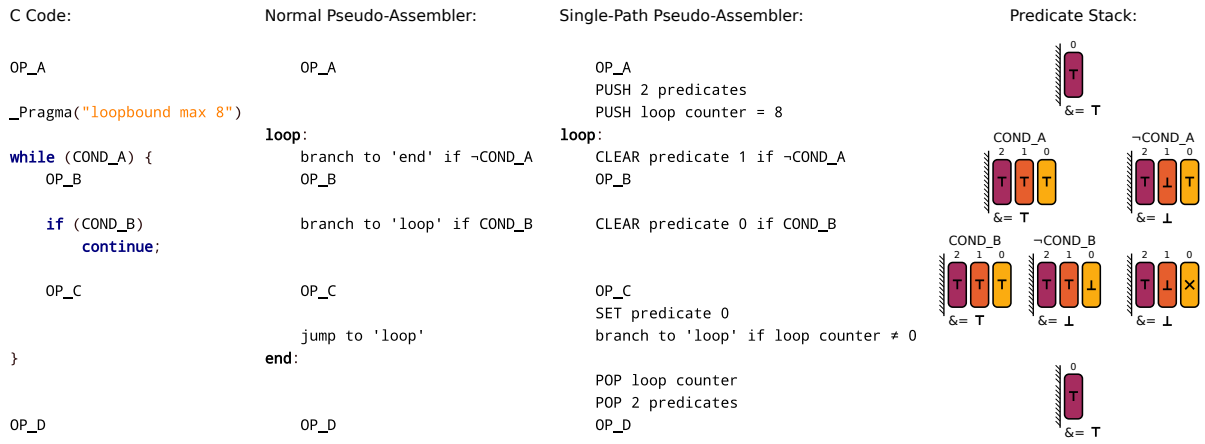

Fig. 3. Example of a loop in single-path code: The loop bound annotation is used to initialize the loop counter in single-path code and the loop is executed for a constant number of iterations. The loop predicate capturing the loop condition and the iteration predicate, which is cleared by a *continue* statement and reset at the start of each iteration, control whether the instructions are actually active.

6 Current Work

We are implementing the single-path generation and the instruction filter for two Reduced Instruction Set Computer (RISC) processors that are synthesized as softcores in a Field-Programmable Gate Array (FPGA):

- LEON3, a SPARC v8 processor core developed by Cobham Gaisler for safety-critical applications.
- IBEX, a RISC-V processor core developed by ETH Zürich and the non-profit organization lowRISC.

Both of these processors have a multi-stage in-order pipeline with predictable timings, thus they execute a given single-path program in constant time regardless of input values.

The single-path code is generated following the method of Prokesch et al. [5], which is applied to the fully compiled and linked executable file generated by a target-dependent version of the GNU C Compiler (GCC).

Currently our implementation allows the successful translation and execution of a set of experimental sample programs. We are working on the completion of the single-path translation and filtering, such that all WCET-analyzable programs can be transformed to single-path and executed on these two processors, and possibly extending the approach to other architectures.

7 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. The predicates are managed on a predicate stack in the filter and are controlled by special instructions that are interpreted by the filter itself. The condition codes of the processor are routed to the filter to allow predicates to be set conditionally.

Single-path code that can be executed by a processor that was upgraded with this filter is generated from regular machine code by applying the single-path transformation method developed by Prokesch et al. to the executable produced by a regular compilation toolchain in a post-processing step. This conversion reorders the basic blocks of the CFG of the program and replaces conditional control-flow branches with special instructions that control the state of predicates. These special instructions are parsed and interpreted by the filter.

We are implementing the single-path generation and the instruction filter for two RISC processors. Currently our implementations allow the successful execution of few simple test programs. We plan to complete the single-path filters for these two processors and potentially implement versions for more architectures. Once fully functional we will evaluate the performance of single-path code on these platforms and compare the WCET of regular code with the constant execution time of the equivalent single-path code.

References

1. Delvai, M., Huber, W., Puschner, P., Steininger, A.: Processor support for temporal predictability - the spear design example. In: 15th Euromicro Conference on Real-Time Systems, 2003. Proceedings. pp. 169–176 (July 2003)
2. Geyer, C.B., Huber, B., Prokesch, D., Puschner, P.: 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). pp. 1–8 (2013)
3. Law, S., Bate, I.: Achieving appropriate test coverage for reliable measurement-based timing analysis. In: 2016 28th Euromicro Conference on Real-Time Systems (ECRTS). pp. 189–199 (July 2016). https://doi.org/10.1109/ECRTS.2016.21
4. Mahlke, S.A., Hank, R.E., McCormick, J.E., August, D.I., Hwu, W.M.W.: A comparison of full and partial predicated execution support for ilp processors. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture. p. 138–150. ISCA '95, Association for Computing Machinery, New York, NY, USA (1995), https://doi.org/10.1145/223982.225965
5. Prokesch, D., Hepp, S., Puschner, P.: A generator for time-predictable code. In: 2015 IEEE 18th International Symposium on Real-Time Distributed Computing. pp. 27–34 (April 2015). https://doi.org/10.1109/ISORC.2015.40
6. Puschner, P.: The single-path approach towards WCET-analysable software. In: IEEE International Conference on Industrial Technology, 2003. vol. 2, pp. 699–704 Vol.2 (Dec 2003). https://doi.org/10.1109/ICIT.2003.1290740
7. Puschner, P.: Experiments with wcet-oriented programming and the single-path architecture. In: 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. pp. 205–210 (2005)
8. Puschner, P.: Transforming Execution-Time Boundable Code into Temporally Predictable Code, pp. 163–172. Springer US, Boston, MA (2002). https://doi.org/10.1007/978-0-387-35599-3_17
9. Puschner, P., Kirner, R., Huber, B., Prokesch, D.: Compiling for time predictability. In: Ortmeier, F., Daniel, P. (eds.) Computer Safety, Reliability, and Security. pp. 382–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
10. Santinelli, L., Guet, F., Morio, J.: Revising measurement-based probabilistic timing analysis. In: 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 199–208 (April 2017)
11. Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P., Rocha, A., Silva, C., Sparsø, J., Tocchi, A.: T-CREST: Time-predictable multi-core architecture for embedded systems. Journal of Systems Architecture **61**(9), 449–471 (2015). https://doi.org/10.1016/j.sysarc.2015.04.002
12. Schoeberl, M., Puschner, P., Kirner, R.: A single-path chip-multiprocessor system. vol. 5860, pp. 47–57 (11 2009). https://doi.org/10.1007/978-3-642-10265-3_5
13. Wenzel, I., Kirner, R., Rieder, B., Puschner, P.: Measurement-based timing analysis. Communications in Computer and Information Science **17**, 430–444 (10 2008). https://doi.org/10.1007/978-3-540-88479-8_30
14. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3) (May 2008). https://doi.org/10.1145/1347375.1347389