A Strategy for Generating Time-Predictable Code

Daniel Prokesch and Peter Puschner

Institute of Computer Engineering
Vienna University of Technology, Austria
{daniel,peter}@vmars.tuwien.ac.at

Abstract. *Prohibiting external control* is one of the key principles engineers apply when building time-predictable computer systems (e.g., time-triggered computer systems do not react to any external interrupts from sensors or devices, but all actions of these computer systems are triggered solely by the progression of the local clock). In this paper we apply this principle of *prohibiting external control* to code generation: The single-path code generator is a compiler that produces real-time code that does not contain any input-dependent control flow. All input-dependent control-flow dependencies are eliminated by if-conversion or by the generation of loops whose iteration counts are fixed. We explain the principle of operation of single-path code generation and illustrate how single-path code generation contributes to the time-predictable behavior of real-time computer systems.

Keywords: real-time computer systems, embedded systems, compilers, code generation, time predictability

1 Introduction

Real-time computer systems are used in safety-critical application domains like the automotive and aerospace domains. In these application domains computer systems must not only deliver functionally correct results. They must also produce these results at the right time. Otherwise a catastrophe like a plane crash might occur. Thus, an important property of each computer system used in a safety-critical real-time application is the temporal correctness of its operation. The worst-case analysis has to identify and analyze all worst-case timing scenarios such that the timely operation of the real-time computer system can be guaranteed for all phases of system operation.

Providing timing guarantees is a highly complex issue, especially as both the hardware and the software used in safety-critical real-time computer systems are getting more complex themselves. To keep systems nonetheless simple, the pre-planning design strategy for so-called time-triggered systems constructs a time schedule for all activities of the computer system at system design time [4, 3]. This means, the points in time when user tasks or operating-system tasks are started or when messages are sent is planned and written into scheduling tables before the system is started. At runtime, the system software of the real-time computer system interprets these tables as time progresses (time-triggered activation), thus strictly controlling the execution of all actions as planned. Such time-triggered systems do not allow for a dynamic change of the plan once the system is in operation, i.e., any external control over the sequencing of actions in the computer system is prohibited.

In this work we take the principle of prohibiting external control on actions in a real-time computer system one step further, from the scheduling level to the code level of single tasks. We present the *single-path code-generation strategy* that compiles C source code to machine code in such a way that the resulting machine code does not contain any input-dependent control flow. The absence of input-dependent control flow makes the execution of the machine code always take the same path through the instructions of the program and thus produces the same instruction trace every time the code is executed (therefore the name *single-path code* [10]).

Within this paper we will explain our approach to single-path code generation. We will first provide more motivation for using single-path code and then present the main idea behind single-path code generation (Section 2). We will then show how the LLVM compiler framework [5] can be extended with a single-path code generator (Section 3). We have run a number of experiments with the extended LLVM compiler framework. In these experiments, we compiled benchmark programs to single-path code and executed the compiled code on the Patmos time predictable processor [14]. These experiments and lessons learned are summarized in Section 4. Following this evaluation, we conclude the paper.

2 The Single-Path Approach

In this section, we would like to introduce the single-path code-generation strategy. In particular, we will answer the questions of (a) *why* one would like to generate and run single-path code and (b) *how* imperative code for real-time systems can be made to execute on the same instruction path for any inputs.

Above, we have motivated the use of single-path code by the fact that removing control-flow alternatives simplifies the worst-case execution-time (WCET) analysis of the generated code. In fact, the generation of single-path code eliminates the task of identifying (in)feasible program paths, one of the main subtasks of WCET analysis [9, 15], from WCET analysis. Besides, there are further advantages of using single-path code. The main advantages are summarized in the following paragraphs.

– The first and main advantage is that single-path code is much *easier to analyze* for its (worst-case) timing than traditional code – analyzing a single stream of instructions has a lower complexity than accounting for the timing of code that allows for a multitude of different instruction sequences.
– Second, if the instruction trace of a piece of code is always the same one can expect *smaller execution-time variations* than for code that executes

different instructions on each execution. This type of execution-time stability is advantageous for control software where a variable latency between inputs and outputs adversely affects control quality.

– Third, single-path code can be used to *thwart certain side-channel security attacks*: if all inputs are processed along the same instruction stream and with identical execution time (e.g., by using a processor with invariable instruction timing, [14]) then attackers cannot exploit observations of the code execution times to draw conclusions about the actual data being processed.
– Finally, precise knowledge about the instruction stream can be beneficial for *speeding up code execution*, e.g., by using the knowledge about the execution path to control the prefetching of code blocks into the fast levels of the memory hierarchy right before they are executed [2].

Making Code Execute on a Single Path

Traditional compilers generate code with input-data dependent branches in the control flow to realize input-data dependent code behavior. Such input-data dependent control flow realizes (a) the branching to conditional or alternative code of *if-then*, *if-then-else* or multi-way branches (e.g., *switch-case*) and (b) loop-exit branches for all types of *loop* statements. A single-path compiler, in contrast, must generate code that executes the same stream of instructions for all inputs. I.e., a single-path compiler has to provide code-generation patterns that bring forth data-invariant control flow for alternatives as well as loop constructs.

The single-path code generation uses the following strategies to generate code for alternatives respectively loop constructs:

Alternative constructs with input-dependent conditions are translated by means of *if-conversion* [1]: Instead of using conditional branches to achieve data-dependent code behavior, the single-path compiler generates predicated code [7], i.e., it serializes the code of input-dependent alternatives and uses predicates to control the activation of instructions and achieve the right code semantics at runtime.

Loops with input-data dependent exit conditions are translated into simple counting loops with a constant iteration count. Thereby, the iteration count of the generated loop is set to the maximum iteration count of the original loop[1]. The exit condition of the original source-code loop is used to compute a predicate for the execution of the loop body of the new loop that is itself translated into predicated code.

Further details about the single-path approach can be found in [10, 11]. The following part of the paper provides details about the realization of the single-path code generation in the LLVM compiler framework.

[1] We assume that the source code is is real-time code for which all loop bounds are known.

3 Generation of Single-Path Code

As a modern state-of-the-art compiler framework, LLVM operates in several phases. The frontend translates the source language to *bitcode*. Most optimizations are operating on this source language- and target-agnostic intermediate representation of LLVM. A backend translates the bitcode to target-specific machine instructions. Because the source code is not translated to machine code directly, the translation schemata described in [11] are not applicable directly. Where in the compilation process can the single-path code generation be integrated?

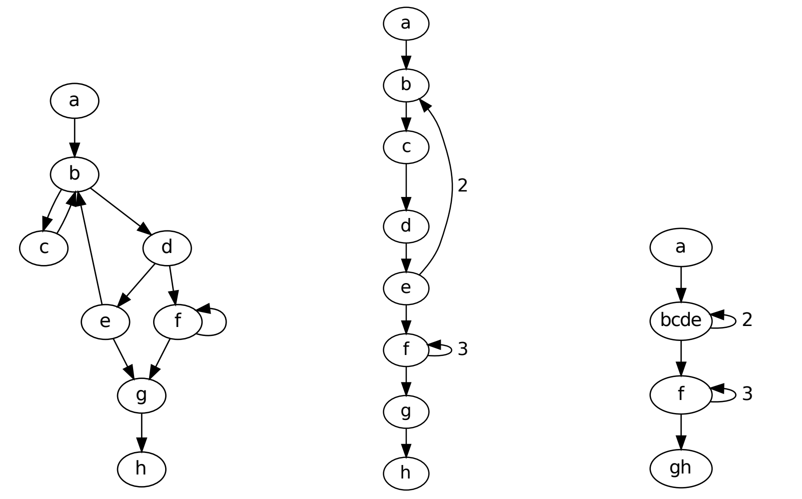
3.1 The Single-Path Graph Transformation

For the Patmos compiler, the single-path code generator is a set of program transformation passes that are executed late in the backend. As such, the problem of generating single-path code requires a suitable formulation on the program representation at that stage. To this end, we have developed the *single-path graph transformation* [8], that operates on the control-flow graph of a given function. Based on the algorithm of Park and Schlansker [7], it transforms the control-flow graph into a graph with linear structure, simple loops, and predicated nodes. It extends the algorithm [7] by transforming any reducible control-flow graph (not only the body of innermost loops) and by producing loops with constant iteration counts. Following the graph structure and the constraints regarding the loop back edges, there exists only a single path through the resulting graph. Furthermore, the transformation involves the insertion of instructions that control the value of the predicates assigned to the nodes.

Predicates are Boolean-valued variables that enable or disable operations. If the predicate is *true*, the operations are performed as usual, if the predicate is *false*, the operations have no effect. In terms of nodes in a control-flow graph, a predicate controls all the instructions of that node. Informally, the single-path graph transformation achieves the following:

> For every valid path in the original control-flow graph, the sequence of nodes on that path is equal to the sequence of nodes on the resulting graph with a predicate value of *true*.

The single-path graph transformation is best illustrated by an example. Figure 1a shows a control-flow graph before the single-path graph transformation. Figure 1b shows the single-path control-flow graph, with constant counts on the loop back-edges. Consider example paths π_1, π_2 in the original control-flow graph in the following table:



(a) Control flow graph (b) Single-path CFG (c) Simplified

Fig. 1: Example illustrating the original control-flow graph and the control-flow graph after the single-path graph transformation.

Original control flow graph	Single-path control flow graph
$\pi_1 = \texttt{abdebcbdegh}$	$\pi_1^{SP} = \underline{\texttt{ab}}\texttt{c}\underline{\texttt{debc}}\texttt{de}\underline{\texttt{b}}\texttt{c}\underline{\texttt{de}}\texttt{ffff}\underline{\texttt{gh}}$
$\pi_2 = \texttt{abcbcbdfffgh}$	$\pi_2^{SP} = \underline{\texttt{abc}}\texttt{de}\underline{\texttt{bc}}\texttt{de}\underline{\texttt{b}}\texttt{c}\underline{\texttt{d}}\texttt{e}\underline{\texttt{fff}}\texttt{f}\underline{\texttt{gh}}$

In the corresponding paths in the single-path control flow graph, π_1^{SP} and π_2^{SP}, respectively, the nodes with predicate value of *true* are underlined and their sequence equals the nodes of the original path. The singleton execution path always contains the same sequence of nodes, albeit with different predicate values.

Details on the single-path graph transformation, including how the predicates are assigned and changed along the execution path, can be found in [8].

3.2 The Patmos Processor

Before elaborating on the compiler passes, we briefly describe the relevant characteristics of the Patmos processor [14] that make it a suitable target for single-path code. Time predictability is the key principle in the design of Patmos. The timing of the instructions of the fully predicated instruction set is independent from the operands (except for latencies originating from the memory hierarchy). Features like dynamic instruction reordering and dynamic branch prediction are avoided in favor of static alternatives. Delays in the in-order dual-issue pipeline are exposed at instruction set architectural level (branch delay slots, load delays). The memory hierarchy is organized as a split cache architecture [13]. The caches are either software managed or at least controllable to obtain a known state, e.g., by flushing the cache contents.

3.3 Compiler Passes for Single-Path Code

As mentioned before, single-path code generation is performed late in the backend. This is due to following reasons. First, LLVM bitcode is SSA-based and predication-oblivious. Although there is support for partial predication in the form of a `select` instruction, which creates a new value as one of two operands depending on a Boolean operand, this form of predication is not sufficient to deal with the difficulties arising from instructions with side effects. Computing the values for alternative paths and discarding the unnecessary ones is only an option when safe values are provided, for example, to memory accesses and division operations in order to prevent access to invalid addresses and division by zero, respectively [6]. The machine instructions of the backend are predication-aware and have predicate operands. Second, the code structure is final at that stage and no instructions are inserted that could invalidate the single-path property. Calls to software arithmetic functions are already visible and the final number of required predicates is known. Third, we can perform optimized register allocation for predicate registers with detailed knowledge of the target, which we explain below.

The compiler passes for generation of single-path code are categorized as (i) preparatory passes, and (ii) the main transformation pass. The preparatory passes include a *unify return* pass, to guarantee that there is only one sink node in the control flow graph of each function, a *lower switch* pass to convert indirect jumps to a cascade of if-else statements, and *function cloning* to restrict single-path functions only to where they are required.

The main transformation pass performs the single-path graph transformation as described in the previous section. After computing predicates for each node in the graph, a specialized predicate register allocator is invoked, which assigns machine registers to the virtual predicates, on basic block (= node) level. Space for spilling predicate registers is allocated on Patmos' stack cache, and the 1-bit registers are stored packed into machine words. Live-ranges of predicates are predominantly nested and cover whole inner loops. This observation is exploited in Patmos to obtain a new set of available predicate registers when a loop is entered by spilling the whole predicate register file, and restoring it when the loop is left. After the assignment of physical registers, the instructions of each block are predicated accordingly. Function calls are executed unconditionally, passing the predicate to the called function. Then, instructions for manipulation and for spilling and restoring of predicate registers are inserted.

Finally, the basic blocks of the transformed control flow-graph are merged wherever possible, as illustrated in Figure 1c. This removal of basic block boundaries leads to a simplified control-flow graph structure with large basic blocks, which gives the final instruction scheduler more opportunities to generate compact and efficient instruction schedules.

4 Experiments

Having a compiler at hand that produces single-path code, we were particularly interested in answering following questions:

– How does the generated single-path code perform in the worst case, compared to conventionally generated code?
– How do latencies caused by the memory hierarchy affect the execution time?

To obtain answers to these questions, we evaluated the single-path code generator on a benchmark based on a real-world application. The *debie1* benchmark is based in the on-board software of the DEBIE-1 satellite instrument for measuring impacts of small space debris or micro-meteoroids, developed by Space Systems Finland Ltd for Patria Aviation Oy.[2]

We generated both conventional code and single-path code for the main tasks of the benchmark. We executed the conventional code to measure the observable range of execution times and additionally applied static analysis. For the measurement, we used `pasim`, the cycle-accurate simulator for Patmos. Each task is executed at least several hundred times in a benchmark run.[3] We used `platin` for static WCET analysis, a toolkit which is part of the compilation tool chain for Patmos [12].

We performed the evaluation with two different hardware configurations:

1. Ideal memory (*ideal*) - Memory accesses do not entail any additional access latency.
2. Ideal data cache (*dcideal*) - Only accesses that go through the data cache do not entail any additional latency. Memory accessed via Patmos' stack cache (2 kB) and method cache (4 kB) exhibits actual memory access latencies.

This choice is motivated by the fact that the serialization of control flow alternatives leads to an increase of the path lengths through the tasks. Masking the impact of the memory hierarchy enables us to quantify this effect solely at the instruction level. By allowing memory access for instructions and call frames, we can evaluate the single-path code in the context of real memory latencies, while maintaining execution-time invariability: On the single execution path, functions are called unconditionally, and space for call frames is allocated on the stack cache for those functions. Hence, every execution has the same sequence

[2] The source code is available at http://www.tidorum.fi/debie1/debie1-e-free.zip
[3] To be more precise, the number of executions of a task is in the range between 394 and 17795.

Task	SP Functions	Predicates	Configuration	Measured	Static Analysis	Single-Path	Ratio
TC_InterruptService	1	55	ideal	[17, 157]	163	306	1.88
			dcideal	[70, 445]	459	696	1.52
TM_InterruptService	1	10	ideal	[27, 38]	47	68	1.45
			dcideal	[78, 132]	141	163	1.16
HandleHitTrigger	3	31	ideal	[44, 7502]	12586	13879	1.10
			dcideal	[106, 7890]	13245	14351	1.08
HandleTelecommand	7	311	ideal	[67, 994]	994	3013	3.03
			dcideal	[179, 1294]	1294	5590	4.32
HandleAcquisition	17	234	ideal	[68, 26878]	29332	35695	1.21
			dcideal	[176, 29985]	36106	39824	1.10

Table 1: Results for the debie1 benchmark.

of accesses to the caches. In addition, we clear the caches before each entry to a single path function to obtain a well defined cache state. As a result, the generated single-path code has a singleton execution time by construction.

4.1 Results

The results of our experiments are shown in Table 1. The column "SP Functions" shows the number of functions that are involved in the tasks' execution and require transformation. This contains the entry function of the task itself and all functions reachable in the call-graph. Column "Predicates" shows the total number of predicates required for the single-path version of the task. This number gives a hint about the breadth of the involved control-flow graphs. The column "Measured" shows the interval [min, max] containing the observed execution times of the conventional variants, while "Static Analysis" shows the WCET bound as computed by `platin`. The execution time of the single-path task code is given in column "Single-Path". Because it is not known whether the actual worst-case path has been observed for the conventionally generated code, though we are primarily interested in worst-case guarantees, we have to consider the statically computed bound for a performance comparison with the single-path code. Column "Ratio" shows the execution time of the single-path code relative to the WCET bound of the conventionally generated code.

For these experiments, we can make some interesting observations. In all cases, the linearization of control flow alternatives leads to an increase in the execution time (bound), hence a ratio greater than 1. The highest ratio was obtained for HandleTelecommand (3.03 for *ideal*). In this particular task, the additional cost stems from serialization of a switch-statement: In the program, a message is read and processed accordingly depending on the message type. In the single-path variant, code for all the different cases is fetched and executed. The effect is even more pronounced when the code is loaded to the method cache (4.32 for *dcideal*). In the other tasks, the original control-flow structure has fewer alternatives in the control flow. As a result, the relative additional cost for loading code from main memory is lower, yielding a lower ratio in the *dcideal* case than in the *ideal* case.

4.2 Lessons learned

Our single path code generator is able to produce code without input-data dependent control flow. Targeting the time-predictable Patmos processor, this code generation strategy further results in code for real-time tasks that not only has a singleton execution path, but also is completely free from execution time jitter, making timing analysis trivial.

This property comes at a cost, as the experiments have shown. The execution time of the single-path code is higher than the statically computed worst-case execution time of the conventionally generated code. This is due to the serialization of control flow alternatives. One way to address the problem is to avoid input-data dependent control flow in the first place. But this has limited use, especially, when one has to deal with legacy code.

Another way would be to incorporate input-data dependence on a higher level of modeling. For example, the HandleTelecommand task of our benchmark performs different actions according to the message type of the incoming message. A type has its particular action, and the set of actions could be considered as *modes* of the task. There is little point in serializing all actions. Instead, by generating single-path code for each action individually, we would obtain a set of execution paths, where each path can be tied to the corresponding action.

5 Conclusion

Single-path code is free from input-data dependent control flow. Our single-path code generator is integrated in the compiler backend for the Patmos processor by adopting the single-path graph transformation. Patmos is a suitable target for single-path code, because it provides a predictable instruction set with full predication and software-controllable caches. Our experiments with the debie1 benchmark have shown that the generated single-path code is competitive with conventionally generated code in terms of worst-case performance, yet it is easier to analyze and exhibits stable execution-time behavior.

As future work we plan to implement compiler optimizations tailored to single-path code, for minimizing the cost introduced by control-flow serialization. Mode-specific single-path code will avoid complete serialization of input-dependent alternatives. It will provide a means to leave branches to mode-specific sections in the code, and the resulting mode-specific execution times could be used in a more differentiated way.

References

1. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: Proc. 10th ACM Symposium on Principles of Programming Languages. pp. 177–189 (Jan 1983)
2. Cilku, B., Prokesch, D., Puschner, P.: A time-predictable instruction-cache architecture that uses prefetching and cache locking. In: Proc. 18th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) Workshops, 11th IEEE/IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS). IEEE CS Press (2015)
3. Kopetz, H., Fohler, G., Grünsteidl, G., Kantz, H., Pospischil, G., Puschner, P., Reisinger, J., Schlatterbeck, R., Schütz, W., Vrchoticky, A., Zainlinger, R.: Real-time system development: The programming model of mars. In: Proc. IEEE International Symposium on Autonomous Decentralized Systems. pp. 190–199 (1993)
4. Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., Schütz, W.: An engineering approach to hard real-time system design. In: Proc. 3rd European Software Engineering Conference. pp. 166–188 (1991)
5. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO'04). pp. 75–88. IEEE Computer Society (2004)
6. Mahlke, S., Hank, R., McCormick, J., August, D., Hwu, W.: A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In: Proc. 22nd International Symposium on Computer Architecture. pp. 138–150 (Jun 1995)
7. Park, J.C., Schlansker, M.: On predicated execution. Tech. rep., Hewlett Packard Software and Systems Laboratory (May 1991)
8. Prokesch, D., Huber, B., Puschner, P.: Towards Automated Generation of Time-Predictable Code. In: Falk, H. (ed.) 14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, 2014, Madrid, Spain. OASIcs, vol. 39, pp. 103–112. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2014)
9. Puschner, P., Burns, A.: A review of worst-case execution-time analysis. Journal of Real-Time Systems 18(2/3), 115–128 (2000)
10. Puschner, P., Burns, A.: Writing temporally predictable code. In: Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. pp. 85–91 (Jan 2002)
11. Puschner, P., Kirner, R., Huber, B., Prokesch, D.: Compiling for time predictability. In: Proc. SAFECOMP 2012 Workshops (LNCS 7613). pp. 382–391. Springer (2012)
12. Puschner, P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and WCET-analysis integration. In: Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013) (2013)

13. Schoeberl, M.: Time-predictable cache organization. In: Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems. pp. 11–16. STFSSD '09, IEEE Computer Society, Washington, DC, USA (2009)
14. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: Towards a time-predictable dual-issue microprocessor: The Patmos approach. In: First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011). pp. 11–20 (March 2011)
15. 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. on Embedded Computing Systems 7(3) (2008)