Informatics

Compiling for Time-Predictability and Performance

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Emad Jacob Maroun
Registration Number 11932440

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Peter Puschner
Second advisor: Prof. Dr. Martin Schoeberl

The dissertation has been reviewed by:

Björn Lisper

Isabelle Puaut

Vienna, July 9, 2024

Emad Jacob Maroun

Erklärung zur Verfassung der Arbeit

Emad Jacob Maroun

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

Wien, 9. Juli 2024

Emad Jacob Maroun

Acknowledgements

First, I thank my advisor, Prof. Peter Puschner, for his guidance and mentoring throughout this work. His open-door policy and openness to listening to my ideas were essential to completing this work. I want to thank my co-advisor, Prof. Martin Schoeberl, for his guidance during this work, for shaping the trajectory of my academic career, and for connecting me with Prof. Puschner.

My doctoral studies were funded and supported by the Doctoral College Resilient Embedded Systems, run jointly by TU Wien's Faculty of Informatics and the UAS Technikum Wien. I thank Prof. Andreas Steininger and Dr. Ulrike Weisz for the opportunity to join the program and their excellent stewardship.

I also want to thank Prof. Isabelle Puaut and Prof. Björn Lisper for reviewing this dissertation and Prof. Wilfried Elmenreich and Prof. Wolfgang Kastner for their roles on the examination committee.

I want to thank all my former Cyber-Physical Systems group colleagues for creating a work environment I enjoyed being part of every day. I want to thank Elisa De Cristo for all the help and effort she has lent me. I thank Andreas Brandstätter for being the perfect office-mate and Dr. Michael Platzer for the great sparring about our shared topics.

I want to thank all the friends I made during my stay in Vienna for giving me unforgettable memories not mentioned in this work. Likewise, I would like to thank my family and friends from home for their constant support and encouragement. Specifically, I would like to thank my father, Jacob Anis Maroun, for always having a bed ready when I needed a break from this work.

Abstract

Single-path is a code-generation technique to disconnect a program's execution time from run-time conditions. Full time-predictability means programs have a constant execution time, which, when coupled with sufficient performance, can replace static worst-case execution time analysis in the design process for a real-time system. This cumulative dissertation collates six works improving the time-predictability and performance of single-path code. Constant execution times are achieved by compensating for varying access times to main memory. Performance is improved by optimizing code that is executed a fixed number of times, by more efficient use of dual-issue processor pipelines, and by enabling the use of generic register allocators for allocating predicate registers. This dissertation also discusses how the challenges of correctness and practicality require this type of research to use the scientific method to advance the state of the art. The selected papers are put into context before evaluating the end-to-end cumulative performance impacts. In addition to achieving constant execution times on a time-predictable processor, the results show varying but significant improvements of up to 145% in performance and a reduced code size of up to 28%.

Contents

CHAPTER 1

Introduction

Safety-critical applications often require hard real-time systems to ensure correct and timely functionality. In applications within medical, automation, or process-control fields, the timely execution of programs is often just as important as their correctness. For example, a self-driving vehicle should be able to scan the road in front of it and identify any obstacles in time to take evasive action to avoid an accident, e.g., by performing emergency braking. The designers of such a vehicle must ensure that it finishes analyzing its surroundings with enough spare time to perform the appropriate actions successfully. The point at which a given program must have its result ready is called a deadline. We call a system *hard real-time* if missing a deadline causes a total system failure. Confidence in the execution time of programs is herein second only to their correct result.

Where performance for non-real-time systems is often measured in average-case execution time (ACET), worst-case execution time (WCET) is a more relevant measure for hard real-time systems. WCET is the longest possible execution time a program will require to finish its task for any input data or runtime environment. It is the value system designers must use when scheduling the various tasks comprising the system to ensure that all tasks can be finished before their respective deadlines. Obtaining the WCET of a program is a non-trivial task. Almost every facet of a system can influence the execution time of a program and its variability. In the program, data-dependent control flow means different program parts are executed depending on what data is received at runtime. The different paths then have different lengths, meaning the final execution time varies with the input data. Modern architectures also introduce variability in the execution time because of their design. Many techniques designed to reduce ACET can result in huge variability in execution time. Caching is a great example, where, on average, it reduces execution times by often negating the need to access main memory. However, it does not address the worst case of the cache not containing the needed data (a *cache miss*), which must, therefore, be fetched from main memory instead. The multitude of sources for execution-time variability makes finding the actual WCET nigh on impossible, and even finding a reasonable estimate for it—a *WCET bound*—often impractical [1, 2].

Compiling programs for hard real-time systems is a unique challenge. When compiling for non-real-time systems, an optimizing compiler will try to reduce the ACET such that, in general, the programs are as fast as possible. For hard real-time systems, average-case performance is of little importance. Instead, optimizations must focus on the WCET. System designers can then more easily schedule all the tasks in the system and provision the cheapest hardware that can still do the job. However, a low WCET is only useful if a designer can measure or estimate it accurately. Specialized code generation and optimization techniques must be used to obtain the lowest possible WCET bound. One such technique is *single-path code generation*, which aims to make finding WCET bounds easier by simplifying the final binary produced by a compiler [3].

While modern architectures and compiling techniques have been very successful at maximizing performance, the time-predictability of the resulting programs has been severely hampered. Full time-predictability is a complete disconnect between a program's execution time and the environment it eventually runs on. Put simply, a program must have constant execution time (CET) regardless of what inputs it gets while running. Full time-predictability is a characteristic that in itself can bring desirable benefits as it is a type of abstraction. Without any execution-time variability, system designers can treat programs like black boxes that do not change if the system around them does. The most obvious benefit is no longer needing to perform WCET analysis; with CET, any execution time is also the worst. Other examples are plentiful and range from side-channel attack resistance to exact battery requirements.

This dissertation explores the question of whether we can unify time-predictability and performance. We first explore how to build on single-path code to achieve CET code. We then explore optimizations specific to single-path code to ensure the performance is practically usable.

This dissertation is organized as follows: The rest of this chapter first discusses the research methods used. Then, the cumulative contributions are summarized, followed by a short description of each paper comprising this work. Lastly, we evaluate the end-to-end impact of all the works combined. Chapters 2 through 7 each contain an embedded paper.

1.1 Research Methods

The research conducted as part of this dissertation is within the area of compiling (specifically in the back-end code generation) and code optimization for real-time systems. There are two central challenges for researchers in this field:

1. **Correctness:** Real-time systems are often used in safety-critical applications like medical devices, control electronics (e.g., planes, vehicles), or production control

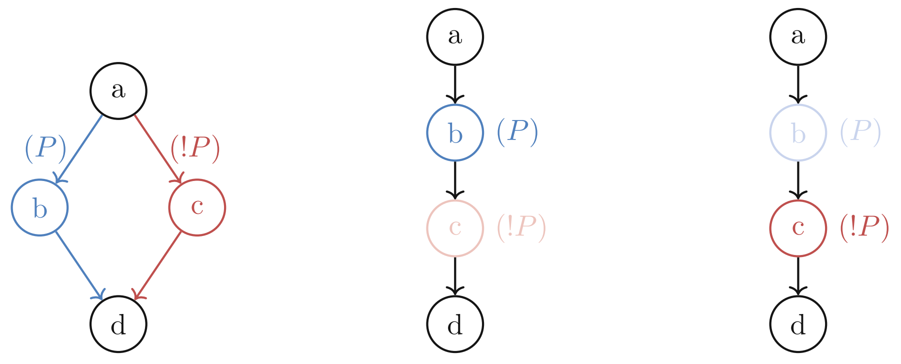
systems. This requires high confidence in the system's behavior under all scenarios. As such, the correctness of the code generated must be maintained when new techniques or optimizations are applied.

2. **Practicality:** Because safety is of the utmost importance, theoretical assurances are not sufficient within this field. Researchers must demonstrate that their proposed techniques or optimizations are practically implementable and work as intended. Practical experimentation is necessary to demonstrate that the solution is implementable in a real system. The closer the experiments are to real applications, the better. Additionally, optimizations must also be shown to have practical benefits. There are several examples of algorithms or optimizations that should provide benefits in theory but that, in practice, are inferior to simpler alternatives with worse theoretical performance. Examples of this can be found in the efforts put into calculating the dominance relation for control-flow graphs. While there are some proposals for linear-time algorithms, it has been a challenge to implement them in a way that is faster than older, non-linear algorithms [4, 5, 6]. Therefore, any optimization must be demonstrated to benefit real-world applications.

The above requirements are exhibited most clearly in the aeronautics industry [7]. Here, the most critical software must be formally validated. However, all planes must still perform real-world flight tests with various performance objectives to ensure that all the systems work together as expected to produce stable flight [8].

The requirements of this dissertation's research area are not generally conducive to using qualitative research methods. We seek to quantify under which circumstances a plane can successfully take off, how fast the control circuitry can react to pilot input, and what probability a given device has of failing. It is an active effort to avoid subjective interference by engineers, pilots, or passengers, meaning a central objective of qualitative methods (investigating the experiences of people [9]) is actively avoided. That is not to say that these methods are not useful within the broader area of real-time systems and aeronautics. For example, the usability of the controls for the pilots and the comfort of the passengers could be a situation where qualitative methods could be utilized [10, 11]. However, this work is so far removed from the end users that we must employ other methods.

The scientific paradigm with its accompanying methods, as described by Eden [12], is most well-suited for this research area. Eden's description of why the rationalist and technocratic paradigms are insufficient for computer science can be seen clearly in this research area. The rationalist's focus on mathematical correctness is not enough for us, as too many unmodeled phenomena may occur in the real world. A counterpoint might be that we need to model more accurately, but then we are constrained by our knowledge of physics, chemistry, and the rest of the natural sciences. The technocrat's focus on testing is also not enough in this area. While we do need thorough testing of the complete systems, many systems are too complex and/or expensive to test exhaustively. For example, a plane can cost hundreds of millions of dollars to build. Exhaustive testing



(a) Traditional (b) $P = true$ (c) $P = false$

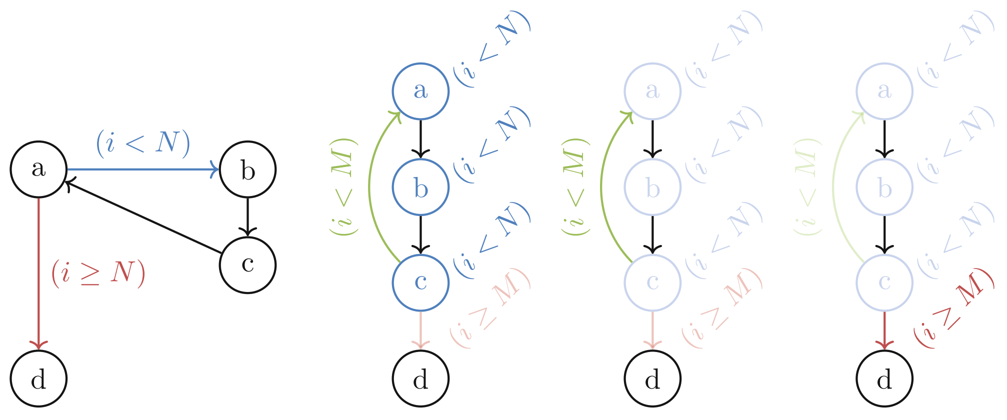
Figure 1.1: If-conversion from branching control flow (a) to predicate-based control flow (b & c).

would need to include testing to failure, which would destroy a plane. Such testing is not economically feasible. Simulation is a technique to alleviate this economic burden, but it has its limitations and often cannot simulate complex systems in a reasonable time. Therefore, we cannot rely solely on a posteriori knowledge about our system.

The scientific paradigm better meets our requirements. The design of a system starts (a priori) with rigorous theoretical foundations for its functionality. This could include formally certifying, e.g., flight software and compilers. When confident in the design, we begin the practical investigations about whether it performs as expected. If there is congruence between the system's performance and our expectations, it gives us confidence that even the parts we cannot practically investigate perform as expected. As Eden describes, this is the scientific method of creating a hypothesis (designing/modeling a system) and running experiments to support or falsify it. This method also works for the optimization of systems. When doing that, we investigate the system's current state and identify things that could be changed to increase performance. We then build the mathematical foundations for performing the optimization. For example, in Chapter 7, we describe how our compiler's single-path transformation executes much unnecessary code. We identify code that is always guaranteed to be executed a fixed number of times and can be more effectively managed in single-path code to avoid unnecessary executions. We then prove (formally) the correctness of the relation and algorithm used to identify the relevant code. We prove (informally) that applying the optimization maintains program semantics and the properties of single-path code. We then implement the optimizations and investigate whether they actually increase performance (and, of course, whether the programs still work as intended.)

1.2 Approach

To avoid the complexities of branching control flow, single-path code generation converts traditional code into code with only one execution path [13]. *Predication* is used to *enable*



(a) Traditional (b) $i < N$ (c) $N < i < M$ (d) $i \geq M$

Figure 1.2: Loop-conversion from branching control flow (a) to predicate-based control flow (b, c, & d).

or *disable* instructions, such that the original semantics are maintained [14]. A predicated instruction takes an additional predicate argument. If the predicate evaluates to true at runtime, the instruction executes normally. However, if the predicate evaluates to false, the instruction's effects are nullified, effectively making it a `no-op`. Disabled instructions take the same time to execute but do not affect any registers or access memory.

If-conversion takes the instructions in the two paths following a branch and puts them in sequence. The instructions are then predicated on the original branch condition, so only the instructions from the needed path are enabled at runtime. In Figure 1.1, we can see the effects of if-conversion on a simple program. Traditional code uses the value of P to decide the branch target at runtime and thus which one of b or c is executed (Figure 1.1a). For converted code, if P is true, b will be enabled at runtime (Figure 1.1b), otherwise c is enabled (Figure 1.1c). *Loop-conversion* alters the iteration logic of loops such that they always iterate the maximum possible amount they would do in the traditional code. The instructions in a loop are then predicated on the loop condition such that they are disabled when the loop would have stopped iterating in the traditional code. The superfluous iterations, therefore, have no effect except maintaining the single-path property. In Figure 1.2, we can see the effects of loop-conversion on a simple loop that iterates up to a runtime value (N) that has a known upper bound (M). The loop condition ($i < N$) traditionally decides if the loop should iterate or exit. After conversion, we can see that the condition keeps the loop body enabled (Figure 1.2b) until the condition becomes false (Figure 1.2c), while the loop is only exited after M iterations (Figure 1.2d). *Function-conversion* alters all function calls to enable them regardless of whether the given path is enabled at runtime. Instead, the functions take an additional predicate argument that predicates their entire bodies. A function called on a traditionally taken path will execute normally with single-path code, but one called on a traditionally skipped path will have no effect with single-path code except maintaining the single-path property by executing its disabled instructions.

When applied to traditional code, the three conversions described cumulatively result in single-path code. However, a naive implementation of these conversions results in performance inefficiencies and does not guarantee CET. We build on the existing work on single-path code to produce code with CET and improving performance through optimization

1.3 Contributions

The cumulative contributions of this dissertation are as follows:

First, we present two techniques to compensate for variations in memory-access times such that single-path code exhibits CET. *Opposite-predicate compensation* (OPC) adds additional memory-accessing instructions with the opposite predicate to that of the original memory-accessing instruction, such that either the original or the compensatory instruction always executes and incurs memory-access latency. *Decrementing-counter compensation* (DCC) tracks how many memory accesses a function performs and adds additional accesses at the end to reach the maximum number of accesses.

Then, we present the *repetition dominance* relation: a variation of traditional dominance that accounts for how many times a node is visited before another. We show how repetition dominance can be used for *pseudo-root optimization*, where functions and basic blocks that are always executed the same number of times can be simplified to need less predication and avoid unnecessary execution. We present proofs of correctness for the properties of the relation, the algorithm that calculates it, and a proof that the optimizations maintain the properties of single-path code.

We discuss how single-path code has unique characteristics that should be accounted for during instruction scheduling. We present a simple yet efficient list scheduler that is predicate-aware. We also explore the effects of allowing additional instruction types in the second issue slot of the Patmos architecture to allow for increased performance without much increase in hardware or software complexity.

Lastly, we present an approach to predicate-register allocation that uses only generic register allocators. We describe a new single-path transformation that uses a two-step register allocation approach, which allows both general-purpose and predicate registers to be allocated efficiently.

1.4 Selected Papers

This dissertation contains six papers published in scientific journals and conference proceedings. The first paper introduces the OPC and DCC techniques for achieving CET using single-path code on systems without intermediate storage. The second paper introduces the constant-loop dominance relation, an earlier version of repetition dominance, which reduces the need for predication and thus increases performance. The third paper explores techniques for bundling instructions for parallel execution on Patmos' dual-issue pipeline. The fourth paper extends the third with an improved bundling algorithm and an extended evaluation. The fifth paper presents a new implementation of single-path code that uses a generic register allocator to allocate predicate registers. The sixth paper extends and improves the work of the preceding papers to maximize performance. It presents the repetition-dominance relation with accompanying formal theorems and proofs. It also explores a different and simpler avenue for instruction bundling using the unique characteristics of single-path code.

My contribution to all the papers started at the planning stage. For each paper, discussions with the co-authors were held to decide on research direction, problem statement design, and solution exploration. After collective agreement on problem statements and proposed solutions, I performed the practical implementation and evaluation. Most of the technical writing was subsequently done by myself, in addition to managing the publication of the final papers and the presentation at conferences where required.

Compiler-Directed Constant Execution Time on Flat Memory Systems

Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner
2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)

The first paper addresses the time-predictability of single-path code running of the T-CREST platform using the Patmos processor. The central issue is that execution-time variability can arise because of the varying latency of accessing main memory. In this paper, we first argue for the value of having programs with CET. We then show how the memory system around a processing core can affect execution times even on systems without intermediate storage like caches or scratchpads.[1] We present automatic compiler techniques for generating CET programs by compensating for the memory-induced variability. The results of the paper show that the best compensation solution uses a mix of the OPC and DCC compensation techniques. They also show that a simpler, hardware-based solution that removes the variability is inferior to the compiler-based solutions presented. However, a mix of hardware- and software-based techniques is likely better than either (but is not explored in this paper.) The results show significant performance and code size impacts for the proposed techniques. Still, there is also potential for comparable performance to the state-of-the-art.

[1] *Flat memory systems* refer to the absence of intermediate storage.

Constant-Loop Dominators for Single-Path Code Optimization

Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner
2023 21st International Workshop on Worst-Case Execution Time Analysis (WCET)

The second paper addresses a significant performance issue for single-path code: the execution of redundant code. The general technique used in single-path code to avoid execution-time variability is to ensure code sequences are executed a fixed number of times. An example is that of loops, where the loop is forced to iterate the maximum possible number of times, with any superfluous iterations being disabled, doing nothing. Likewise, all function calls are executed, even when the given path is not needed at runtime. In both cases, code is executed that does nothing except require execution time, ensuring all program runs maintain the same execution time. This paper reduces the amount of redundant code execution by identifying code that is always guaranteed to be executed a fixed number of times (in the original, non-single-path program.) This code is guaranteed never to induce execution-time variability, meaning single-path code does not need to ensure it does so by forcing its superfluous execution. The paper introduces the *constant-loop dominance* relation on control-flow graphs (CFGs) that can recognize basic blocks executed a fixed number of times. The relation is used to identify *pseudo-root* functions—functions that are guaranteed to be called a fixed number of times—which can be optimized to avoid unnecessary code execution. The results show sporadic but significant performance improvements for single-path code with minor effects on the total binary code size.

Towards Dual-Issue Single-Path Code

Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner
2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)

To maximize performance without sacrificing predictability, Patmos implements a *dual-issue pipeline*, which allows instructions to be *bundled* in pairs that will be executed in parallel. The compiler must explicitly output bundled instruction pairs to use the dual-issue pipeline. A unique characteristic of single-path code is significantly increased instruction-level parallelism (ILP), which can be easily exploited using a dual-issue pipeline. This paper is the first attempt at supporting instruction bundling for single-path code. It presents a single-path code generator that uses the control flow to recognize pairs of basic blocks where only one is enabled simultaneously. Such pairs can then easily be bundled by putting the first block's instruction in the first issue slot and the second block's in the second issue slot. The proposed technique is general and can be extended to use different algorithms for finding block pairs and how to bundle them exactly. Synthetic benchmarks show that the presented techniques have a great potential for performance improvements. However, this does not translate well to more representative workloads, with only minor improvements when using standardized benchmarks.

Compiling for Time-Predictability with Dual-Issue Single-Path Code

Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner
Journal of Systems Architecture, Volume 118, 2021

This paper is an extension of the previous paper. It slightly changes the specific bundling algorithm to be used as a proof-of-concept to handle more complex cases, e.g., nested if/else statements. It also expands the evaluation to include further synthetic benchmarks to explore the potential of bundling. The evaluation also includes additional benchmark programs and a real-world drone control program. The results show only minor improvements compared to the previous paper, meaning the potential of using the second issue slot of Patmos is still unrealized.

Two-Step Register Allocation for Implementing Single-Path Code

Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner
2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)

Predicate registers introduce additional challenges to register allocation. Because they decide whether instructions have an effect, allocations must be predicate-aware to take advantage of the knowledge. Generic register allocators cannot do this. The *single-path transformation* is a collection of passes added to the compiler that manage the conversion to single-path code. This paper presents a new and improved single-path transformation that exclusively uses generic register allocators. It splits the allocation into two steps, the first for general-purpose registers and the second for predicate registers. The bulk of the single-path transformation happens between the two, giving maximum flexibility for both types of registers to be allocated. Results show that the efficiency of register allocation in the new transformation is greatly improved, with examples of several orders of magnitude less data movement between registers and the stack frame. This translates to an improvement in both performance and code size.

Predictable and Optimized Single-Path Code for Predicated Processors

Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner
Journal of Systems Architecture, Volume 154, 2024

The last paper is a summary and improvement of the themes that were addressed by the previous papers. First, the constant-loop dominance relation lacked a formalized definition and a proof of correctness for the algorithm that calculates it. Indeed, further effort showed that the relation was incorrect in some edge cases, and the algorithm was, therefore, not provable. This paper presents the *repetition-dominance* relation, which is a variation of the constant-loop dominance relation that addresses the latter's issues. Lemmas and theorems are presented, showing specific properties of the relation, accompanied by proofs. The properties are then used to prove the correctness of the algorithm for calculating the repetition-dominance relation. Similarly to the first paper, we use the relation to reduce the execution of redundant code.

This paper also takes a second stab at enabling the use of Patmos' dual-issue pipeline. Here, we recognize that single-path code has characteristics that make it uniquely amenable to instruction scheduling, which is the act of reordering instructions to optimize performance. First, we recognize that single-path code produces few but large basic blocks containing instructions with a high level of ILP. Simple scheduling techniques should perform well when applied to individual basic blocks. This contrasts with the previous papers on scheduling by bundling after the single-path transformation has finished instead of before it. The paper also explores further instruction scheduling opportunities unique to single-path code that allow additional instructions to be bundled, even where traditionally impossible.

The paper's results show that for the majority of benchmark programs, the performance is competitive to that of the state of the art. Additionally, a few cases were observed where the performance of the CET single-path code was superior to that of traditionally WCET-analyzed code.

1.5 End-to-End Evaluation

This dissertation covers many techniques for optimizing the performance of single-path code. The following chapters evaluate the impact of these techniques individually. Therefore, it would be interesting to see the cumulative effects of all the techniques on performance and code size.

A proper evaluation of the work's performance impact would need to use WCET-analyzed single-path code without the memory-access compensation techniques described in Chapter 2. However, the Platin analyzer used for WCET analysis of Patmos code often encounters bugs when analyzing single-path code. This means we can only get a WCET bound for a small subset of our benchmark programs, rendering any comparison useless. As a substitute, we will use CET single-path code with the memory access compensation technique (OPC) as the baseline, as that is the simplest implementation of CET and likely the least optimal. The baseline will also disable all the optimizations developed during this work, use only a single issue-slot, and use the original single-path transformation. We compare the baseline to CET single-path code using the improved single-path transformation, both issue-slots, all the optimizations, and the *Hybrid* memory compensation technique using the `comp4` compensation function.

We see our work's performance and code-size impact in Figure 1.3.[2] The blue bars show the raw performance impact, i.e., comparing the execution times of the baseline to the optimized code ($\frac{baseline-optimized}{optimized} \times 100$). For most programs, the improvement is less than 25%; however, with multiple examples of enormous improvements of up to 145 % in

[2]We use the same programs as in Chapter 6, excluding the `duff` program.

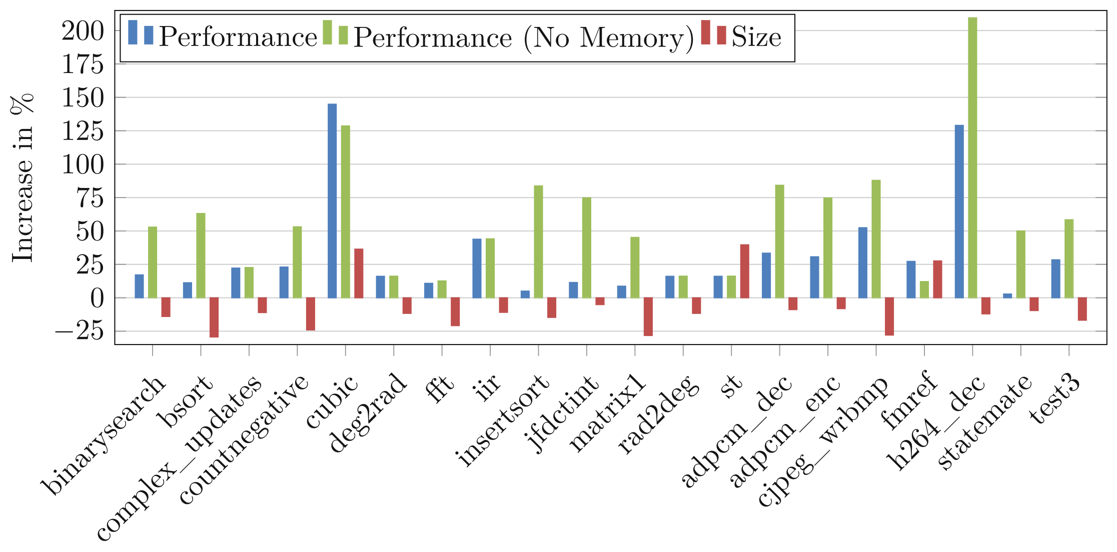


Figure 1.3: Cumulative performance and code size increase of enabling all techniques and optimizations covered in this dissertation.

the case of `cubic`. Looking at the green bars, we see the performance improvement when main-memory-access stall cycles are subtracted from the execution time. In this case, the performance impact is more significant, with most programs seeing close to or more than a 50% increase. The maximum increase now also becomes 210 % for `h264_dec`.

It is relevant to show the impact of the optimizations without main-memory-access stalls because the current work does not incorporate hierarchical memory for random data accesses. For example, using scratchpads likely significantly reduces execution times while maintaining CETs. However, the caches that are used (stack and method caches) also trigger main memory accesses, meaning removing all these stalls is not representative of the performance of implementations with hierarchical memory for random data accesses. Therefore, the green bars are a best-case scenario, with the actual performance of practical and mature single-path implementations likely between the blue and green bars.

Lastly, the red bars show code size increases of using the techniques in this dissertation. Code size is significantly impacted, with mostly reductions of up to 28 % and with some examples of increases of up to 40 %. As described in Chapter 7, the increases in code size can be attributed to the implementation of memory-access compensation code and can likely be reduced significantly with smarter decisions on when to duplicate functions.

Cumulatively, the work described in this dissertation significantly and positively affects single-path code performance. Improvements are especially impressive in the programs that most exhibit pathologically high overhead when using single-path code. However, as covered in Chapter 7, more work must be done to reduce cases with pathologically poor performance.

CHAPTER 2

Compiler-Directed Constant Execution Time on Flat Memory Systems

Abstract—*Time predictability is a central requirement for real-time systems. The correct behavior of such a system can only be achieved if the results of programs are ready in time to affect the environment. Execution times of modern systems can vary for many reasons, meaning complex analyses must be performed to ensure that the execution time is bounded and that a task always finishes before its deadline. Care must also be taken to ensure that nefarious actors do not exploit the varying execution time to compromise the system's integrity. Avoiding variable execution times can greatly simplify systems, is inherently more secure, and eliminates the need for complex analyses.*
In this paper, we first argue for the value of having programs with constant execution times. We then show how the memory system around a processing core can affect execution times even on systems without intermediate storage like caches or scratchpads. We present automatic compiler techniques for generating constant execution time programs and evaluate their implementation on the Patmos architecture. We show that combining our two compensation techniques is generally superior to either on their own. We compare the performance of our implementation to the estimates produced by the Platin worst-case execution time analyzer. While our implementation significantly impacts performance, it is generally manageable and has the potential for comparable execution times.

2.1 Introduction

For real-time systems, the timely execution of a program is just as important as achieving correct results. Execution times can vary considerably from one run of a program to the next. There are many sources for this variability, from those inherent in the program flow to the specific state of the hardware at the beginning of execution. The complexity of modern systems makes it nearly impossible to deduce exactly how long a given program will run. Worst-case execution time (WCET) analysis is used to argue about execution times by finding an upper bound. However, with increased program and hardware complexity, finding reasonable WCET bounds is often impractical [1, 2].

Single-path code is a code generation technique that enforces a single execution path [13]. With no input-data-dependent control-flow, the same instruction stream is executed every time, making WCET analysis easier to run and more accurate.

As part of the T-CREST project [15], the Patmos instruction set architecture was specifically designed to support single-path code [16]. All its instructions are predicated, which allows them to be disabled at runtime. The compiler can then use if-conversion, among other techniques, to turn control-flow decisions into straight-line code [17].

While single-path code makes executions much more predictable, it does not remove all sources of execution time variability. When different code paths access memory unevenly, variability is introduced from the different amounts of access latency incurred. Single-path code does not manage this variability automatically.

Constant execution time (CET) is a characteristic that can bring desirable benefits. Without any execution time variability, system designers can treat programs like black boxes that do not change if the system around them does. CET is also more secure as a task does not leak information through its execution time. The most obvious benefit of CET is no longer needing to perform WCET analysis; any execution time is also the WCET.

This paper presents changes to the single-path code transformation to ensure that generated programs have CET. The result is code with a CET when run on a system with no memory hierarchy (a flat memory system) but is extensible to more complex memories.

The contributions of this paper are: (1) an argument for the value of having CET programs, (2) a description of how to correct for memory access variability with two specific techniques, and (3) an implementation of an automatic CET code generator in a compiler for Patmos.

The paper is organized into eight sections: The following section presents related work. Section 2.3 argues for the value of having CET. Section 2.4 provides background on the Patmos processor and single-path code generation. Section 2.5 discusses in which contexts single-path code lacks CET and how to compensate for the variability. Section 2.6 describes how we have implemented the variability compensation in the compiler.

Section 2.7 evaluates how our compensation techniques perform and how our execution times compare to traditionally analyzed code. Section 2.8 concludes.

2.2 Related Work

Single-path code is introduced in [13] as a code generation technique specifically for real-time systems. The authors of [17] present an algorithm for automatically generating single-path code from WCET analyzable source code. They show that the generated code can run on the Patmos architecture and that the conversion to single-path has a significant but manageable performance cost. The authors of [18] compare the performance of the Patmos processor using single-path code (running on an FPGA at 80 MHz) to that of an off-the-shelf ARM processor using traditional code (running at 1.4 GHz). Their results show single-path code being competitive with the observed WCET of the ARM processor for an implementation of a quadcopter control algorithm. In [19], we investigate the performance impact of adding support for dual-issue pipelines to single-path code. We show how single-path code has the potential to make efficient use of the additional pipeline because of its high instruction-level parallelism. While single-path code is significantly easier to analyze, it does not outright eliminate the need for analysis. In this paper, we build on the preceding works such that the resulting code has CET on architectures with flat memory.

An architecture must have dedicated support for single-path code. At a minimum, this must include a conditional move instruction, but can also be achieved with fully predicated instructions—which Patmos has. This requirement precludes existing architectures from using single-path code if they do not have the required features. In [20], the authors present an instruction filter that can be added to an existing processor that does not support single-path code. The filter then alters the instruction stream on-the-fly to produce the same effect as single-path code. It does so by monitoring the state of the processor and control-flow and issuing instructions that are effectively single-path. To avoid memory access variability, the filter ensures that load and store instructions are still issued even when disabled, throwing out the result. This is similar to our work, except we eliminate memory access variability through software instead of hardware. We also show the benefit of having the option to not querying memory.

WCET bounds are needed to perform task scheduling and schedulability analysis for a system. The field is well-studied with textbooks and surveys focusing on various aspects of scheduling [21, 22]. Both WCET bounds and CET can often be used for scheduling or schedulability analysis. However, this is only sometimes the case. The authors of [23] analyze the best-case response time of tasks on multi-core systems that use inter-processor communication. Their technique needs access to bounds on both WCET and best-case execution time. However, when CET is used, the technique gets simplified since the two values become identical, rendering jitter inconsequential. On the other hand, the authors of [24] propose a scheduling algorithm that can reclaim unused execution time (when tasks terminate earlier than their WCET) to increase schedule performance. This would not work for CET (or at least have no beneficial effect), as the tasks would always use all of their allotted time.

Because WCET bounds can be hard to obtain, some work has gone into designing real-time systems without good bounds on execution time. In [25], the authors present a technique for schedulability analysis given ranges of WCET bounds. The analysis can be used in the system-design stage before any programs are fully implemented and WCET is estimated instead of analyzed. The results can then guide the implementation to reach an acceptable WCET. The authors of [26] devise a technique for task scheduling that does not need WCET bounds to be known at the system design stage. Instead, the scheduler gets live execution-time feedback as the system runs and corrects the schedule accordingly.

To make WCET analysis easier, the authors of [27] present a multi-core architecture that can run in a *WCET Computation Mode*. In this mode, the hardware will run with additional delays in the communication bus. These delays ensure that a program always experiences a latency larger than what it will experience during actual operation. While this work is similar to what we present in this paper, it neither guarantees CET nor eliminates the need for WCET analysis. Instead, it is meant to be used with a measurement-based WCET analysis technique.

The logical execution time (LET) paradigm centers around removing the variability of output times [28]. It dictates that a program must always output its results at a specific time after it reads its inputs. While our work does not provide this guarantee, it is compatible with LET and can make it easier to implement since it removes the need for a WCET analyzer (which is otherwise needed with LET.)

2.3 The Case for Constant Execution Times

It is generally accepted that, in the face of varying execution times, WCET bounds can be used as a substitute for actual execution times. In this section, we will argue for the value of programs having CET. Our arguments have three themes: simplicity, security, and availability.

2.3.1 Simplicity

For real-time systems, execution times are relevant, as they are used to design a complete system with tasks that meet their deadline constraints. When execution times are variable, a system's design and modeling must consider this variability, which increases complexity. This complexity inevitably propagates, as we see clearly in the literature on task schedulability.

Two useful properties of homogeneous multiprocessor scheduling algorithms are affected by task execution times: predictability and sustainability [21]. A scheduling algorithm is *predictable* if a reduced execution time cannot result in a longer response time for a job. An algorithm is *sustainable* if and only if, given a schedulable task-set, having that task set perform better would also keep it schedulable. Reduced task execution times are one way of "performing better." While predictability has been shown to hold for all priority-driven, pre-emptive scheduling algorithms [29], the same does not hold for all dynamic priority algorithms. Unless predictability is specifically considered or proven for such algorithms, they cannot be considered useful [21]. When using programs with CET, predictability will be an implicit property of all scheduling algorithms, as no run of a program can have a reduced execution time. Likewise, it simplifies the requirements on sustainability since execution times cannot decrease.

When simplifying requirements, the space of possible solutions to a problem increases. For example, an algorithm is given in [30], which can determine an optimal schedule for jobs with known arrival and execution times. Such an algorithm is automatically disqualified if execution times are variable, even when job arrival times are known. Indeed, it has been shown that, in some cases, optimal scheduling is only possible by knowing the exact execution times [31].

2.3.2 Security

In the same sense as time-predictability being a non-functional requirement for real-time systems, security is often a concern that must be accounted for in applications where real-time systems are used [32]. As embedded systems are increasingly included in and control important aspects of modern life and are ever more connected to the internet, guaranteeing correct behavior becomes less feasible without designing for security. Real-world examples already exist of such systems being targeted or susceptible to intrusion [33, 34, 35].

A good example of how security must be part of the basic design of a system is encryption. While most encryption standards are mathematically vetted to be secure, they are only as secure as the system they run on [36]. Many aspects of modern systems can be used to obtain data from a running system that may compromise the integrity of a security scheme. One such aspect is the variability of execution times, which can be exploited to systematically extract secrets from running processes by measuring their execution times.

While CET is recognized as a protection against timing attacks [32, 37], CET is not generally feasible if a system is not designed for it. Therefore, massive efforts are spent trying to find reasonable alternatives or approximations. Examples include encryption implementations using techniques similar to single-path code on the source code [38, 39, 40], compiler techniques also similar to single-path code [41], and systems that can manage the variability by injecting noise into the execution times to make attacks less fruitful [37].

Varying execution times are a security risk. Any system design that accepts the variability must spend significant effort ensuring the required level of security is maintained. The inherent security of CET should not be disregarded and only be given up for significant gains that outweigh the effort potentially needed to plug the security hole.

2.3.3 Availability

To acquire WCET bounds, a dedicated tool that supports a given processor must be used: a static analyzer. Such tools are not trivial to develop or maintain. They must accurately model the target and employ sophisticated analysis techniques to achieve accurate bounds. Most notably, analyzers must model individual processor implementations as timings often vary from one implementation to the next. As such, the availability of a good analyzer is a non-trivial issue that must be handled whenever a new processor is developed or used. This contrasts with our work, which only needs to be implemented in the compiler and will work on any processor/system that adheres to the requirements.

Even when an analyzer is available, performing the analysis adds another non-trivial step to the development cycle, which adds time and costs to a project. Additionally, the analyzer might not be able to produce a WCET bound within an acceptable time frame as the size of the analyzed programs increases [42].

An excellent example of the difficulties associated with analyzers is the T-CREST platform's analyzer Platin [43]. It was initially developed in cooperation with a commercial company. It interfaced with their proprietary tool that provided advanced techniques like cache analysis. However, as the original project ended, the proprietary tool stopped supporting the Patmos instruction set. Therefore, the accuracy of the bounds Platin now produces is severely lowered, mainly because *all* data-cache accesses are assumed to miss.

In contrast to the maintenance of an analyzer, implementing CET only involves adding passes to the compiler. This can be done as an extension of the compiler implementation, involves the same skills as developing the compiler in the first place, and does not require the compiler to model the architecture in depth. Additionally, acquiring the execution time is as simple as running the program on the target platform with arbitrary input data.

Based on our three arguments, CET should be preferred as the default design for real-time systems. Varying execution times should only be accepted where they specifically increase value or solve a problem that CETs cannot. A cost-benefit analysis should ensure that the system-wide increase in complexity, the potential decrease in security, and the increase in the developer workload are worth it.

2.4 Background

2.4.1 The Patmos Processor

Because Patmos was explicitly designed for real-time systems, the Patmos instruction-set architecture (ISA) is a RISC-style architecture with features that make it time predictable and are optimized for lower WCET [16]. It has an in-order, dual-issue pipeline that maximizes throughput while being highly predictable. It is part of the T-CREST platform, which is also designed for real-time systems using predictable on-chip communication and memory controllers [44].

All instructions are predicated by one of eight boolean predicate registers, i.e., they take the predicate register as an operand. If the value of the predicate is true, the instruction is *enabled*, which means it executes normally. If the predicate is false, the instruction is *disabled*. It still gets executed in the same amount of time. However, it does not update any registers, nor does it—crucially for the work of this paper—read from or write to any memories. Effectively, it becomes a *no-op*.

Another important feature of Patmos is its split caches. It has a method cache instead of an instruction cache [45], which allows the compiler to decide which blocks of code or whole functions should be loaded into the cache at a specific point. This means method-cache misses only occur at designated times, e.g., at function calls and returns. The traditional data cache is split into a stack cache and a data cache [46]. The stack cache is also compiler-controlled and stores the call stack and function-local data. Patmos uses typed load and store instructions. There are dedicated instructions for accessing data through the data cache, stack cache, or directly from main memory. The function prologue and epilogue include instructions to spill and restore the relevant stack-frame data, meaning usage of it in the rest of the function never causes a cache miss. Using the method and stack caches in single-path code means no execution time variability is present from loading instructions or accessing data through the stack cache. Notably, spilling and restoring registers can never cause stack-cache misses and, by extension, does not contribute to variability. This allows us to ignore any instructions targeting the stack in our implementation while maintaining the correctness of our techniques. As we will describe later, the remaining variability comes from the interplay between predicated instructions and accessing data through the data cache or directly from main memory.

Patmos is paired with a memory-tree network-on-chip on the T-CREST platform that uses time-division multiplexing arbitration [47]. This network is designed for analyzability, with each memory access by a Patmos core being serviced within an upper bound. When only one processing core is present in a T-CREST configuration, the accesses are serviced in constant time. Constant latency is an absolute requirement for our techniques to work. However, if multiple cores are configured, loads may be serviced in a varying number of cycles, negating our techniques. Therefore, we do not address multi-core use cases in this paper.

2.4.2 Single-Path Code

Single-path code was initially intended to make analyzing the WCET of code computationally easier. Single-path code uses predication to convert the branching control flow of a function into an instruction stream with only one path. This removes all data-dependent execution-time variability from the instruction stream, making it an obvious starting point for achieving CET. To convert a program into single-path code, we need to use three techniques:

If-Conversion

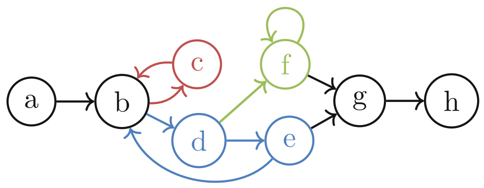
First, we need to convert any conditional branching into a sequence of predicated instructions, such that only the needed path's instructions are enabled at runtime. The resulting code always executes all instructions in both paths. However, only one path's instructions are enabled at any given time. Looking at Figure 2.1, we can see the result of transforming a program to single-path code. The basic block `b` conditionally branches to either `c` or `d`. The color coding of Figure 2.1a's blocks matches the conditions that led to that path being taken. In Figure 2.1b, the colors indicate that only if the corresponding condition is true will the block's instructions be enabled at runtime. As such, we can see how if-conversion results in `b` always leading to first `c` and then `d`. However, only if the red condition holds at runtime will `c`'s instructions be enabled. The same holds for `d`, leading to either `e` or `f` in the traditional code, but eventually leading to both in the single-path version. Notice how we have not colored the edges in the single-path version, as they are always taken.

Loop-Conversion

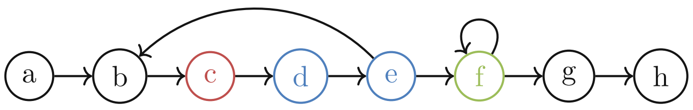
Loops may iterate a variable number of times depending on runtime conditions. To avoid this variability, single-path code converts loops such that they always iterate the maximum possible number of times. Any superfluous iterations are instead disabled using predication. The minimum and maximum iteration counts are usually given in the source code as an annotation on the loop. Looking at our example, we can see the program has two loops, one containing the blocks `b`, `c`, `d`, and `e`, and the second containing only `f`. The single-path loop maintains a count of how many iterations have been executed and keeps looping until the maximum is reached. Inside the loop, the condition that traditionally breaks out of the loop is instead used as the predicate to all the instructions. This condition will become false at some point, meaning any further iterations will have their instructions disabled.

Function-Conversion

Lastly, single-path code also has to account for function calls. Say we have two branching paths, one of which performs a function call while the other does not. If we predicate the function call as we do for the rest of the instructions, it will not cause control to shift to the called function. This means the function's instructions are not executed (neither enabled nor disabled) if the path it was called from was disabled. Instead, function-conversion copies all functions that are called within single-path code. The copies are then edited to take an extra predicate register argument, which specifies whether the function was called from an enabled or disabled path. The body of the copied function is then predicated on that register, meaning if it is called from a disabled function, it will be disabled, too, and vice versa. The call instruction in the caller is not predicated, instead being provided with the predicate of the calling code to pass on. This ensures all functions are always called and executed, regardless of whether their callers are enabled or disabled.



(a) Traditional



(b) Single-Path

Figure 2.1: Conversion of a program with branching control flow (above) to single-path (below).

2.4.3 Definitions

Basic Block: A sequence of instructions whose execution always starts at the first instruction and may only branch on the last. This paper usually uses the term *block*.

Control-Flow Graph (CFG): A directed graph of blocks where the edges model how control flows from one block to the next. A branch is modeled as a block with two outgoing edges in the CFG—one for each path.

Dominate: A block dominates another block if all paths leading to the latter must first go through the former.

Loop Header: In a loop, the header block is the one that dominates all the other blocks in the loop, i.e., it is the entry to the loop. We associate every block in the CFG with the header of the innermost loop containing it. We also treat the whole function as a pseudo loop, where the initial block of the function is the header. Therefore, all blocks in the function have a header.

Back Edge: A back edge has its source block in a loop, with the target block being the header of the same loop.

Exit: An exit edge has its source block in a loop with the target block outside the loop. Informally, the edge exits the loop. The source block of an exit edge is an exit block.

Forward CFG (FCFG): An acyclic CFG resulting from removing all back edges from a CFG. Additionally, all the FCFGs used in this paper will replace any inner loops with their header and construct a separate FCFG for the inner loops. Exit edges from the inner loop are converted to edges outgoing from the header to the same target block. As such, any function's CFG is converted to multiple FCFGs—one for each loop—that refer to each other by their headers.

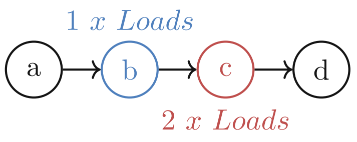


Figure 2.2: Single-path program with uneven memory accesses.

2.5 Achieving Constant Execution Time

As we mentioned in Section 2.4, even though single-path code has a single instruction stream, it does not mean execution times are constant. Any system with multi-cycle memory access latency will exhibit varying execution times for single-path code. This also assumes that the given processor does not prompt the memory system on disabled memory-accessing instructions. T-CREST is such a system. The Patmos ISA specifies a 1-cycle latency for loads (meaning the loaded value is unavailable in the cycle following the load). However, the platform has a 21-cycle main-memory access latency.

This section will describe why variability in memory accesses causes varying execution times. We will then describe the principles that can compensate for memory access variability, followed by presenting two specific compensation techniques that we implemented in our compiler: Opposite-predicate compensation (OPC) and decrementing-counter compensation (DCC). Afterward, we will briefly discuss the security implications of our work.

2.5.1 Memory Access Variability

Take the single-path program in Figure 2.2: It branches at `a` to either `b` or `c` and converges again on `d`. The code in `b` issues a single load instruction while `c` issues two load instructions. In its single-path form, only one of `b` or `c`'s instructions are enabled at a time. While this does not result in variability for most instructions—since their execution time is identical whether enabled or disabled—this is not the case for instructions accessing memory. For example, enabled loads must query the memory system for the data. However, when disabled, no querying is done. While the load still takes time to run through the pipeline, no stalls are issued waiting for the memory system to return data. Thus, enabled loads (and other memory access instructions) incur a higher latency than disabled ones. On the default implementation of T-CREST on the DE2-115 FPGA board, this latency is 21 cycles.

Execution time variability is introduced if different paths through the program issue different numbers of memory accesses. In our example program, `b` only issues one load while `c` issues two. In a run of the program where `b` is enabled, memory access latency is incurred only once. When `c` is enabled, the latency is incurred twice. On the T-CREST platform, that will result in a 21-cycle difference in execution time, depending on which path is enabled. Had the difference in the number of loads been bigger, the execution

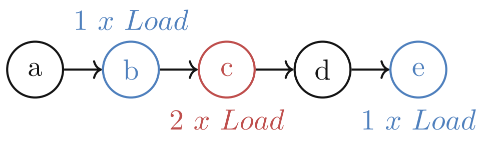


Figure 2.3: Single-path program with corrected memory accesses.

time difference would be proportional to it; for every additional load, the difference increases by 21.

While our example uses only direct accesses to main memory, the variability is also exhibited in hierarchical memory systems. Caches or scratchpads often incur multi-cycle latencies. E.g., on T-CREST, accessing the shared scratchpad can incur a 13-cycle latency [48]. With caches, we also have additional latency from cache misses, compounding the variability.

2.5.2 Variability Compensation

To achieve CET, we must compensate for the uneven number of memory accesses in different program paths. This can be done in many ways. However, the general idea is to add instructions to the program that add execution time where needed. E.g., this could be done by adding a set of predicated *no-op* instructions that are only enabled when their respective memory access is disabled.

However, the general strategy we will use is the insertion of superfluous memory accesses, i.e., load and store instructions. This allows us to compensate for a given disabled memory instruction by executing another—enabled—compensatory memory instruction. In the general case, we would have to match each type of memory access instruction to the same type of compensatory instruction, i.e., loads must compensate for loads and stores for stores. However, loads and stores have the same latency on the T-CREST platform. This means we can use loads to compensate for stores too. Therefore, we will not distinguish between loads and stores in the rest of the paper.

All compensations will be done by loads using the `r0` register as both the source and destination. We can do that since address 0 is always valid to load from, and `r0` is a read-only register, meaning no data is changed in memory or registers by the compensatory loads. Figure 2.3 shows a possible compensation for the original program in Figure 2.2. `e` is added with the same predicate as `b` and has one compensatory load. The result is that both paths through the program (initially either through `b` or `c`) now have two loads and, therefore, the same execution time.

We will now present two specific compensation techniques we implemented in our compiler. They perform differently depending on how much variability is in a given function. Both ensure that regardless of what happens at runtime, the same number of memory accesses are performed by a program.

Opposite-Predicate Compensation

The first variability compensation technique we present is designed to be simple to implement and have minimal overhead for low-variability code. The algorithm goes through all the instructions in a function, and whenever it encounters a memory access, it adds a compensatory load with the same predicate but negated. The result is that either the original instruction or the compensatory load is always enabled, meaning the memory accesses latency is always incurred once for each of the original memory instructions.

While this technique is easy to implement, it can also add significant overhead. Memory access latency is incurred for every memory instruction on all paths. It also adds many new instructions to the program, one new load for every existing memory instruction, meaning additional cycles are wasted on top of all the extra latency incurred. However, if a function has few memory-accessing instructions, this technique's overhead should be negligible.

Decrementing-Counter Compensation

The following variability compensation technique is designed to amortize the compensation overhead. The key is remembering that to achieve CET, we need all paths through a function to have the same number of enabled accesses to memory. The number of accesses we need equals the number of accesses the path with the most accesses has. By analyzing the function, we can count how many accesses are on each path and find the path with the most accesses. We will designate its number of accesses as A_{max}. At the end of the function, a loop can perform extra accesses, depending on which path was taken, such that the final number of enabled accesses becomes constant (A_{max}).

Throughout the function at runtime, we will maintain a counter, `c`, which at the end of the function will contain the number of additional memory accesses we need to perform to reach A_{max}. `c` is initialized to A_{max} at the start of the function and decremented throughout. We use the edges between blocks to track where decrements must be inserted. Initially, we label each edge with the number of accesses its source block performs. To account for the accesses in the final block, we add its access count to all incoming edges. At this point, if we add a decrement in each source block matching the labels, we would have a working compensation. However, minimizing the number of added decrement instructions through constant propagation optimization will maximize performance. This compensation is only based on the control flow and does not depend on the single-path transformation. Additionally, the technique requires us to use a register to hold the counter, which lives throughout the function. Therefore, it is best to add the compensation before register allocation, while the single-path transformation happens after. This differs from OPC, which needs to know which predicates are assigned to each instruction by the single-path transformation and, therefore, must be performed after it.

2.5.3 Security of Compensation

While we asserted earlier that CET is inherently more secure against timing attacks, it is not a blanket security guarantee. There are other ways of attacking a system than just timing its end-to-end execution. E.g., memory-bus contention or energy use can be monitored. This implies that not all compensation algorithms are equally secure in different environments. In our case, DCC does many memory operations at the end of each function. As such, it might be less secure than OPC if used on a system where monitoring memory contention is possible. The choice of compensation technique must synergize with the application requirements and system choice. Alternative techniques to the ones presented in this paper might be needed for specific use cases.

2.6 Implementation

We extend the open-source work presented in [17] to produce CET single-path code. Patmos' compiler is based on the LLVM compiler framework [49]. Its frontend, called Clang, produces LLVM intermediate representation called Bitcode. Bitcode is then compiled by the backend into machine code. The previous work and our extensions all reside in the backend. LLVM also provides Compiler-RT, which is a set of runtime functions. The compiler may assume that these functions are always present and are usually linked into any final program. This includes functions implementing functionality that is not natively available on the machine. For Patmos, this includes division and floating point operations. In the rest of this section, we highlight the changes we made to the single-path transformation to accommodate our techniques and achieve CET.

2.6.1 Data-Cache Elimination

During instruction selection, the compiler automatically inserts instructions that access non-stack data through the data cache. To ensure the data cache is not used in our implementation, we convert all such loads/store instructions into their counterparts that access the same data directly from main memory. This ensures we can focus on these instructions moving forward.

2.6.2 Compensation Selection

While the code still uses virtual registers (i.e., before register allocation), we must decide which compensation technique each function should use. The user can supply the compiler with a flag dictating whether all functions should use OPC, all DCC, or "hybrid." The third option allows the compiler to select which of the two compensation techniques to use for each function.

The selection starts by analyzing the function to find A_{max} and its counterpart A_{min}: the least number of memory accesses any path may execute. This analysis is a simple path enumeration: We first count the number of memory-accessing instructions in each basic block. Then we walk the CFG summing up access counts and tracking the minimum and maximum accesses. When two paths merge, we update the tracking accordingly. We also scale accesses to the loop bounds such that blocks after loops have the tracking account for the highest and lowest number of iterations the loop will take. We chose this approach for its simple implementation. More complex techniques can be used to reduce compile time, but that is outside the scope of this paper.

If $A_{max} = A_{min}$, all paths access memory the same number of times with no need for compensation. Otherwise, a restricted DCC is used to figure out how many instructions would at least be needed to implement it. This is a lower estimate, as depending on register allocation, more or fewer instructions may actually be needed. Then, the main-memory accessing instructions are counted, as the same number of instructions would be needed to be added by OPC. The heuristic we use to select between the two simply chooses the lowest one. This ensures that functions with few accesses do not incur the overhead of DCC's compensation loop.

2.6.3 Decrementing-Counter Compensation

Any function that has been assigned DCC is immediately transformed. However, because the code is in LLVM's machine code representation at this point, the regular, built-in optimization passes cannot be used. We are therefore forced to implement a simplified constant propagation optimization:

We run our compensation on the FCFGs of the function and optimize the decrements as follows: Starting from the header, we forward each edge's label to the edges outgoing from our edge's target block. E.g., say we have blocks `a`, `b`, and `c`, with 3, 6, and 2 loads respectively. `a` $\rightarrow$ `b` is first labeled with 3 and `b` $\rightarrow$ `c` with 6. Then, `a` $\rightarrow$ `b`'s 3 is forwarded to `b` $\rightarrow$ `c` resulting in it being labeled with 9. We then continue forwarding the 9 to any of `c`'s outgoing edges. If a block has multiple outgoing edges, they each get forwarded the accesses from the incoming edge. We cannot forward the accesses if a block has multiple incoming edges, as each might be different. Instead, we stop forwarding, so we decrement the counter in the predecessors by the amount labeled. If we encounter a back edge, we always stop the forwarding. This ensures that each loop iteration decrements the counter by the number of accesses needed. Nothing special is done for the exit edges except to note that the label value will be used when labeling in the outer loop. Remember that inner loops are only represented by their header in the outer loops. Any exit edges are represented by edges from the header to a block in the current loop. When encountering a header, we forward accesses from incoming edges to the outgoing as usual.

In Figure 2.4, we have annotated the edges with the result of running DCC on our example program. For simplicity, we assume each block performs only one memory access. We also assume the `b`-loop may iterate up to three times. The `f`-loop may iterate five times. This means the program will perform at most 16 memory accesses. That happens when the first loop iterates maximally, exiting through `d` $\rightarrow$ `f`, which also loops maximally before exiting to `g`. Our technique results in six decrements being needed:

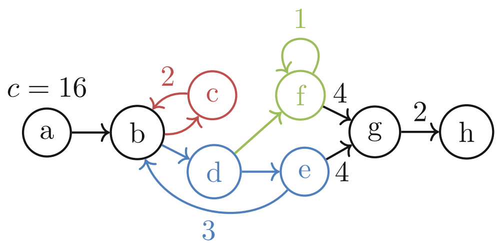


Figure 2.4: Program using the decrementing counter compensation. Assumes the `b`-loop iterates up to 3 times and `f`-loop up to 5 times.

The first loop's two back edges `c` → `b` and `e` → `b`, the second loop's back edge `f` → `f`, the two edges converging on `g`, and lastly `g` → `h`. We have labeled each edge with how much the counter should be decremented. Notice how the decrement for `e` → `g` is based on the one accesses from $a \rightarrow b$, which is forwarded to `e` → `g` (since the first loop is only represented by `b` in the outer-most FCFG) plus the three accesses from `b` to `e`. Notice also how both edges to `g` should decrement by 4. This coincidence can be optimized by forwarding the decrements to the `g` → `h` edge, saving two instructions. However, say `f` had a different number of accesses, these two decrements would differ and could not be forwarded.

After the optimization, we are ready to insert the compensation code. First, the initial block assigns A_{max} to a virtual register. This is the counter. Then, we add instructions that decrement the counter at the source block of any labeled edge. Lastly, we need to insert the compensation loop in the final block of the function. To optimize performance, we do not add the loop directly into the function. We instead add a special compensation function to Compiler-RT that performs the compensation. This function takes two arguments: the number of memory accesses it needs to compensate for and the maximum number of accesses it may be asked to compensate for $(A_{max} - A_{min})$. The function is then a simple single-path loop that performs a number of enabled loads equal to the first argument and uses the second argument for loop conversion, such that the function always runs in constant time. In the calling function, we insert a call to the compensation function in the final block. We use this implementation because the compensation function is called from many single-path functions. This means its code should often be available in the method cache, making it more efficient to call than to have every function include the code in-line.

2.6.4 Opposite-Predicate Compensation

OPC needs to know which predicates are assigned to each block and instruction. Therefore, the pass implementing it must run after the pass in the single-path transformation that makes predicate assignments (called *Single-Path Info* in [17]). Then, we simply go through all instructions and add our compensation loads with the negated predicates.

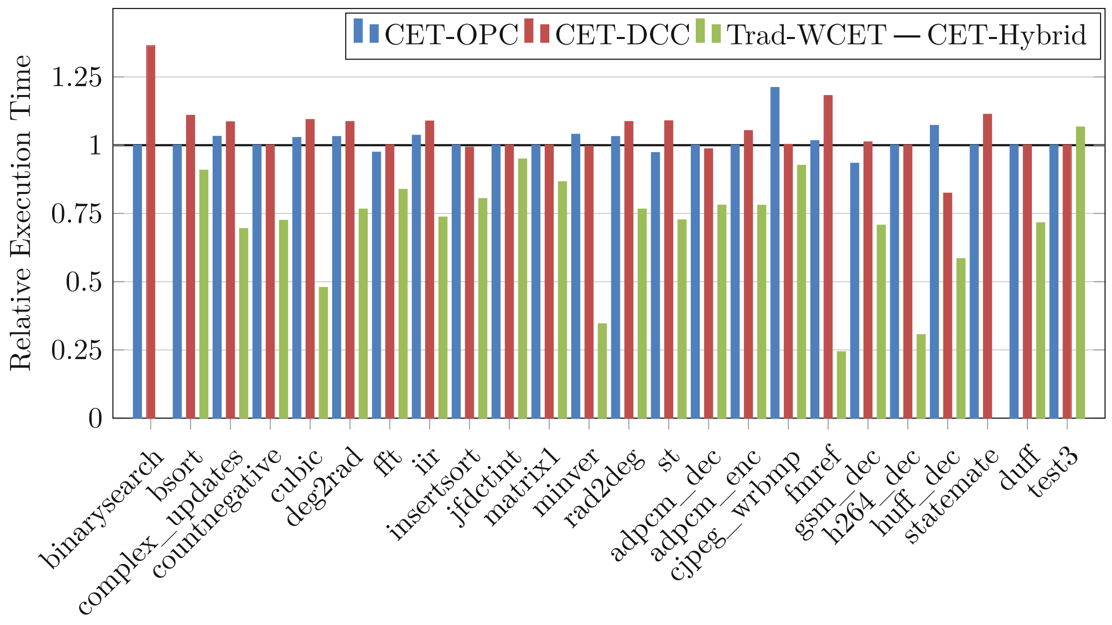


Figure 2.5: Execution time comparison for each configuration. `CET-Hybrid`'s execution cycles is the baseline at 1.

2.7 Evaluation

We use a subset of the TACLe benchmark suite to evaluate our work [50]. We include only those programs that successfully compile for all our different compilation configurations and produce the correct result when run.[1] We excluded programs for the following reasons: 8 programs had recursive calls, and 4 had loops without valid bounds. WCET-analyzable code can have neither recursion nor unbounded loops, so we disregard programs with it. 3 programs were compiled wrong for traditional code. 18 programs either compiled wrong or threw an error when producing single-path code without CET. These are due to various bugs in the single-path implementation. Two programs failed to compile when using DCC due to a bug. We also excluded the `filterbank` program because all the CET configurations saturated the Patmos simulator's cycle counter. However, Platin produced a WCET bound of 4 160 152 660. Seeing as the simulator's cycle counter is 32-bit (max value: 4 294 967 295), it is difficult to say much about the execution time differences in this program. In the end, we have 24 of the programs to look at.

2.7.1 Compensation Techniques

To get a sense of how our two compensation techniques for achieving CET perform, we compare the execution time of the benchmark programs using three different compilation configurations: (1) Using OPC for all functions (`CET-OPC`), (2) using DCC for all functions

[1]All compilations use Clang's `O2` optimization level, the highest our implementation supports.

Table 2.1: Number of functions assigned each compensation technique when using `CET-Hybrid` and how many instructions each added to the program.

	Functions			Instructions	
	None	OPC	DCC	OPC	DCC
binarysearch	0	1	0	2	0
bsort	0	1	0	4	0
complex_updates	10	6	7	20	142
cubic	15	16	16	58	380
deg2rad	10	9	11	30	202
fft	0	0	1	0	16
iir	10	6	7	20	142
insertsort	0	1	0	11	0
minver	5	1	33	28	390
rad2deg	10	9	11	30	202
st	16	16	16	58	380
adpcm_dec	2	1	0	3	0
adpcm_enc	1	2	0	5	0
cjpeg_wrbmp	2	1	0	5	0
fmref	24	31	48	136	789
gsm_dec	0	2	1	8	69
huff_dec	0	1	1	13	57
statemate	0	1	0	33	0
test3	101	0	0	0	0

(`CET-DCC`), and (3) using the hybrid approach described in Section 2.6 (`CET-Hybrid`). Figure 2.5 shows the relative execution times in cycles, where `CET-Hybrid` is the baseline at 1. A bar higher than 1 signifies that the configuration requires more cycles to execute. A bar lower than 1 means it requires fewer cycles to execute. E.g., for `binarysearch`, `CET-OPC` requires the same number of cycles as `CET-Hybrid` (both 667), while `CET-DCC` requires more cycles (910).

The first thing to note is that `CET-OPC` and `CET-Hybrid` have the same execution time for 12 of the programs. E.g., `countnegative` and `jfdctint`. Looking at Table 2.1, we can see the compilation data of `CET-Hybrid`. The three left columns show the distribution of functions that used OPC or DCC in each program or, under *None*, no compensation at all because no variation in the memory accesses exists. We have excluded programs that need no compensation from the table. We confirm that the 12 programs had all their functions needing no compensation or using OPC exclusively. For most programs, `CET-Hybrid` was the most performant configuration (or at least equal to the others.) Therefore, it seems `CET-Hybrid`'s simple heuristic that chooses between OPC and DCC is mostly acceptable. However, there is room for improvement since we can see that, e.g., `gsm_dec`'s `CET-Hybrid` execution is higher than `CET-OPC`.

The heuristic chooses one function to use DCC, and that is clearly sub-optimal. Likewise, `huff_dec`'s `CET-Hybrid` configuration is considerably worse than `CET-DCC`, meaning the function assigned OPC would have been better served with DCC.

In the right two columns of Table 2.1, we can see the distribution of how many instructions each of OPC and DCC added in total to the program. Overall, the 12 OPC programs have slight variations in their memory accesses. E.g., `jfdctint` inserted 0 instructions; no variation exists, and no compensation is required. Six programs in total required no compensation. `bsort`, on the other hand, does have some variation, requiring four instructions to be added by OPC. If we look at the programs that used a mix of OPC and DCC, we can see that roughly the same number of functions are compensated with each, with `minver` being a heavy outlier requiring mostly DCC. This confirms that it is beneficial to have a compensation technique with minimal overhead for cases where little compensation is needed. We can also see that DCC tends to add many more instructions than OPC. Remember that our heuristic only chooses DCC if it requires fewer instructions to be added than OPC. Therefore, DCC's numbers are an approximate lower bound on how many instructions OPC would have added. Thus, a compensation technique geared more towards high-variability functions also benefits performance. We can also confirm this in Figure 2.5, as the programs where `CET-Hybrid` used a mix of OPC and DCC mostly have a lower execution time than the two other configurations.

Our results show that a minimal-overhead compensation technique is critical to performance. However, `CET-OPC` does add significant overhead by introducing new load instructions. It is possible to avoid this overhead if the hardware can assist. While Patmos does not support this, we can imagine having a mode-switch that dictates whether disabled memory accesses query main memory. This would allow functions assigned `CET-OPC` to enable that mode instead of adding extra loads. However, `CET-DCC` functions would disable it to avoid incurring the main memory latency too many times. Our results show that a core that always triggers memory access latency on disabled instructions (with no facility for the program to control it) would perform worse, as `CET-DCC` would not be available for high-variability functions or functions where many paths execute few memory-accesses each. Take `cjpeg_wrbmp`, where `CET-Hybrid` provides a 21 % speedup over `CET-OPC`. Looking at the runtime statistics from the simulator, we see that `CET-OPC` performed 16 702 requests to main memory. Since each request is performed by one of two memory instructions (either the original or the compensation), at most the same number of cycles could be saved by hardware assistance.[2] This still leaves a speedup of 16 % for `CET-Hybrid` with no hardware assistance.

2.7.2 DCC Compensation Function

Since DCC uses a compensation function at the end of all other functions, it is interesting to look at how its implementation affects performance. To investigate that, we

[2]We did not account for method cache behavior since `CET-OPC` only added five compensatory loads, which is negligible compared to the program's total method-cache use of 787 bytes.

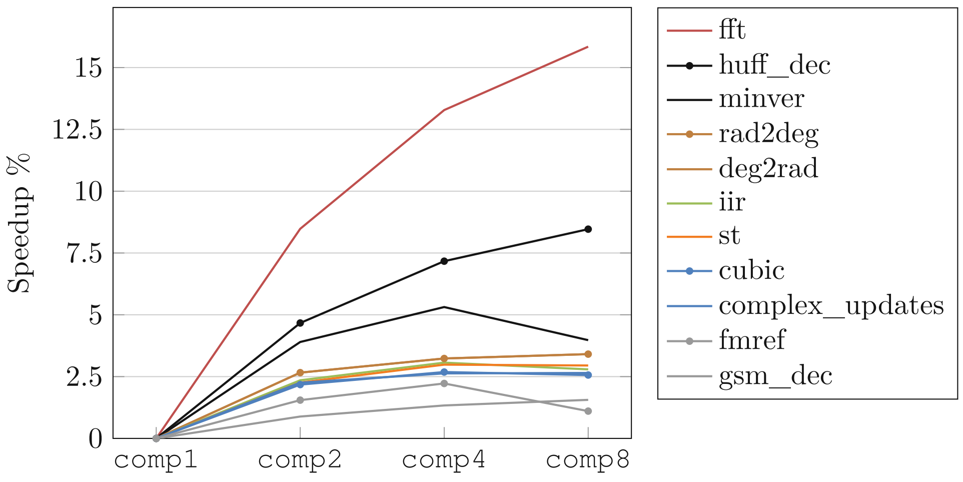


Figure 2.6: Speedups gained by using different compensation functions in DCC. The speedups are relative to using the `comp1` compensation function.

implemented four different compensation functions. They are all primarily comprised of a single-path loop, where either 1, 2, 4, or 8 compensatory loads are performed in each iteration. We will refer to these functions as `comp1`, `comp2`, `comp4`, and `comp8`, respectively. As we increase the number of loads per iteration, performance should increase as it amortizes the overhead of the looping. E.g., `comp1` requires 6 instructions per iteration while `comp4` requires 9. However, to ensure all functions can handle non-power-of-two compensation amounts, extra code outside the loops must be added to handle the remainder. `comp2` only requires two additional instructions for this, `comp4` requires 9, while `comp8` has a full copy of `comp2` following its primary loop, as that requires fewer instructions than inlining it all (10 versus 18 instructions). The instruction counts for our four compensation functions are 9, 13, 22, and 29, respectively. Note that the execution times in Figure 2.5 all use `comp2`.

In Figure 2.6, we see the speedup gained by using the different compensation functions. We include the programs whose `CET-Hybrid` used DCC for at least one function. We can see that most functions benefit from "bigger" compensation functions. The most significant improvement happens between `comp1` and `comp2`, where we see gains from roughly 2.5 % to 7.5 %. Another less than 1 % is then generally gained by using `comp4`, while a few gain considerably more, e.g., `fft`. Using `comp8` brings marginal additional speedups for most, with outliers either gaining significantly more (`fft` for a total gain of 16 %) or seeing reduced performance (`minver` and `fmref`). There may be multiple reasons for this. First, since all other functions call the compensation functions, they are often needed in the method cache. Therefore, the larger they are, the bigger the chance they force more cache evictions for the other functions. Additionally, the performance of our different compensation functions varies based on the exact maximum compensation needed for each single-path function. E.g., `comp8` has the worst performance at each exact power of 8 because it is needlessly iterating in its secondary loop.

2.7.3 Comparison to WCET Bounds

Since CET can be used in place of WCET bounds produced by an analyzer, it is interesting to see how our work performs compared to the Platin analyzer. We include the WCET bounds produced by Platin in Figure 2.5 with the name `Trad-WCET`. The first thing to note is the lack of a bound for `binarysearch` and `statemate`. Platin produced a bound of 262 for `binarysearch`. However, actually running the program requires 337 cycles. Clearly, Platin produced a bound that was too low. For `statemate`, Platin threw an error. This reinforces our assertion that availability is not trivial for WCET bounds. Not only can the analyzer fail to produce a WCET bound, but we must also ensure that its bounds are equal to or higher than the actual WCET.

Looking at the rest of the WCET bounds, the general trend is that they are lower than our best configuration, some more than others. However, `test3` has a lower execution time using CET than the bound. This shows that CET has the potential for comparable performance in some cases. Especially where single-path's weaknesses are less prevalent. `test3` is such a case, as it has minimal branching and fixed-iteration loops. For most of our benchmarks, CET has a 10-40 % higher execution time, with some pathological cases significantly higher. The worst is `fmref` with `CET-Hybrid` taking about 4 times longer than the bound to execute.

We are optimistic about these results. First, for most cases, the performance reduction of CET over WCET bounds is manageable. We can imagine an application where a performance penalty of, e.g., 20 % would be acceptable in exchange for improved security, reduced time-to-market, or a faster development cycle. Additionally, single-path code still has many optimization opportunities that can result in substantial performance benefits. As an example, this work includes an optimization that allows single-path code and our compensation techniques to more accurately see what code is always run or only conditionally run. Going into detail about this optimization is out of the scope of this paper. However, in the case of `cubic`, this optimization alone provides a 98 % speedup for the `CET-Hybrid`. We are confident that further optimizations can significantly improve performance and reduce extreme penalties for CET code.

2.7.4 Source Access

Patmos and the T-CREST platform are available as open-source and include the contributions of this paper. The Patmos homepage can be found at http://patmos.compute.dtu.dk/ and provides a link to the Patmos Reference Handbook [51], which includes build-instructions.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-llvm-project.[3]

[3]Commit Hash: 19dfe015c8c225fe34b88f8aaa4bcdbbf87f95dc

2.8 Conclusion

In this paper, we present techniques for generating programs with CET. We first laid out the case for the value of CET: Being conceptually simple, CET is easier to work with and enables additional possibilities when designing a system; it is inherently more secure against timing attacks; and it is easier to develop and maintain than WCET-analysis tooling. We then argued for why single-path code does not guarantee CET and how to compensate for variable memory accesses. We gave two specific memory access compensation techniques: OPC is best used for simpler functions with low variability in accesses, while DCC maximizes performance when variability is high. Together, these two techniques can be used to achieve CET programs. Lastly, we compared the execution times of programs using our techniques with the WCET bounds produced by the Platin analyzer. We showed that our implementation of CET generally resulted in an increase of 10-40 % in execution time compared to the WCET bounds. While we also showed pathological cases with significantly increased relative execution times for CET, we also saw cases with comparable execution times. Against our expectations, we even saw a case where CET outperformed the analyzer.

Our work is limited to simple systems. It does not support intermediate storage (e.g., scratchpads) or data caches. However, our techniques can be extended to account for some limitations. E.g., adding support for scratchpads should be easy; we treat it as another memory area and compensate accesses using the techniques presented in this paper. Our techniques also do not work with multi-core systems with varying memory access latency (like multi-core T-CREST.) However, any memory system with a guaranteed upper bound on access latency can be converted to have constant latency by adding a delay mechanism that ensures the maximum latency is always observed (e.g., by using a counter.) It should therefore be simple to convert any WCET-analyzable multi-core memory system to work with our techniques.

CHAPTER 3

Constant-Loop Dominators for Single-Path Code Optimization

Abstract—*Single-path code is a code generation technique specifically designed for real-time systems. It guarantees that programs execute the same instruction sequence regardless of runtime conditions. Single-path code uses loop bounds to ensure all loops iterate a fixed number of times equal to their upper loop bound. When the lower and upper bounds are equal, the loop must iterate the same number of times, which we call a constant loop.*
In this paper, we present the constant-loop dominance relation on control-flow graphs. It is a variation of the traditional dominance relation that considers constant loops to find basic blocks that are always executed the same number of times. Using this relation, we present an optimization that reduces the code needed to manage single-path code. Our evaluation shows significant performance improvements, with one example of up to 90%, with mostly minor effects on code size..

3.1 Introduction

Real-time systems are unique in their timing requirements. In addition to producing the correct logical results, real-time programs must produce these results within a specific time frame called the *deadline*. A result produced after the deadline is unacceptable, regardless of its logical correctness. Real-time systems must statically guarantee that a task terminates within the deadline. Here, a program's *worst-case execution time* (WCET) is the critical metric. In the simplest case, if the WCET can always be shown to be shorter than the deadline, we know that the program will always produce its result in time for it to be useful. For multi-task and multi-processor systems, scheduling must be done using each task's WCET to ensure all tasks adhere to their deadlines.

It is almost impossible to know the actual WCET of a program. Therefore, WCET analysis provides an upper bound for it. This *WCET bound* can be used instead of the real WCET when designing the system and verifying its timings. However, the halting problem has shown that creating a program (in this case, a WCET analyzer) that can tell whether any given program will terminate is impossible [52]. To get around this inconvenience, real-time programs are developed with certain restrictions that allow the code to be analyzable without running afoul of the halting problem. One such restriction is to have loops with a bound on the maximum number of iterations. In a real-time program, all loops must have an upper bound on the number of iterations they may perform at runtime. This is a guarantee that the programmer provides — — often in the form of an annotation in the code—to the WCET analyzer, which allows the analyzer to calculate an upper bound to the execution time. A *best-case execution time* (BCET) is also often of interest for task scheduling [23]. To enable efficient BCET analysis, a lower bound on loops is also often provided such that the analyzer does not have to use zero as the default lower bound. If a loop's lower and upper bounds are equal, we call it a *constant loop*; a loop that always executes the same number of iterations.

Single-path code generation is a code-generation technique that ensures that programs execute the same sequence of instructions regardless of runtime conditions [13]. This type of code makes WCET analysis much easier, as the analyzer does not have to account for the program executing different code traces based on what happens at runtime. The properties of single-path code can significantly affect execution time [17]. Therefore, it must be optimized to reduce the execution-time overhead.

The control-flow graph (CFG) is a directed graph that shows how execution can flow through a function.[1] Each node represents a block of sequential code, with edges specifying where execution can continue. We use *block* and *node* interchangeably in this paper. If a node in the CFG has multiple outgoing edges, we call that a branch. Depending on some runtime condition, execution continues at the target node of one edge. Loops in a function are represented by cycles in the CFG. The dominance relation identifies whether a node is guaranteed to be executed before another node. This relation is critical in

[1]We do not consider inter-procedural CFGs in this paper.

compiler construction to ensure correct code generation and optimization [53]. However, the relation does not account for loop bounds and constant loops.

In this paper, we present a new CFG relation called *constant-loop dominance*. It is a variation of dominance that accounts for whether loops are constant to find blocks executed a fixed number of times. Functions called from such blocks can be optimized to reduce the overhead of converting them to single-path code. The contributions of this paper are: (1) a definition of the constant-loop dominance relation and an algorithm for calculating it; (2) a description of an optimization to single-path code that makes use of the relation; and (3) an implementation of the algorithm and optimization in a compiler that produces single-path code.

The paper is organized into six sections: The following section presents related work. Section 3.3 provides background information to support the understanding of the rest of the paper. Section 3.4 introduces the constant-loop dominance relation, an algorithm for finding constant-loop dominators, how it is used to optimize single-path code, and a brief description of the implementation. Section 3.5 evaluates our optimization's performance and code size impacts. Section 3.6 concludes the paper.

3.2 Related Work

The dominance relation is fundamental within compiler construction. Its first description was given with a simple, $\mathcal{O}(n^4)$ algorithm [54]. It was used to implement global common expression elimination and loop identification. Its use continued in other important advances like enabling the efficient computation of static single assignment form [55], which opens up further optimization opportunities [56, 57, 58]. Significant work has been put into reducing the runtime complexity of computing the dominance relation [59, 60, 61]. The state-of-the-art includes an algorithm that runs in $\mathcal{O}(m\alpha(m,n))$, where n is the number of nodes, m is the number of edges, and α is the inverse Ackermann's function [62]. Finally, the quest for a linear time algorithm has resulted in several proposals [6, 5, 4]. The challenge has been translating the theoretical runtime complexity into practical implementations that outperform the older, non-linear algorithms.

Knowledge about loop bounds is a fundamental requirement for analyzing WCET. As such, any annotation language must include the ability to specify bounds [63]. However, since manual annotations can be tedious for programmers to provide and be a source of imprecision and errors, significant effort has gone into automatic methods for finding loop bounds [64, 65, 66]. Effort has been put into finding scenarios that can automatically derive loop bounds. E.g., upper loop bounds can be derived by assuming a loop terminates and then enumerating the state-space of the variables that influence the loop exit [67]. Machine learning has also been used to try and find loop bounds [68]. While our work only uses annotated loop bounds to find constant loops, any method for finding loop bounds is compatible.

Single-path code was introduced as a code generation technique specifically for real-time systems [13]. It can be automatically generated from any WCET analyzable source code, with a significant but manageable performance cost [17]. Single-path code is challenged by its execution-time overhead. One avenue for improving this is to take advantage of its inherent instruction-level parallelism when scheduling on a VLIW architecture [19]. In [69], we extend single-path code with techniques to compensate for execution-time variability from memory accesses. This ensures that single-path code has a constant execution time, eliminating the need for WCET analysis.

3.3 Background

3.3.1 The Patmos Processor

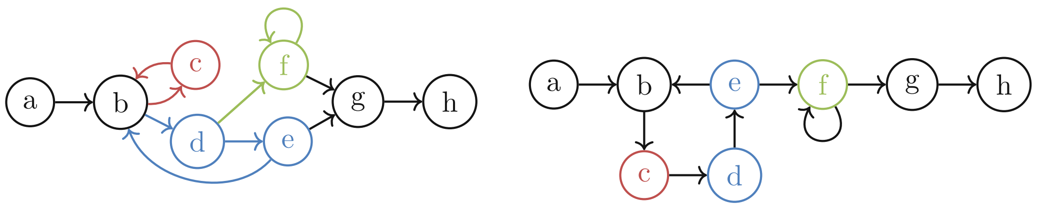
Patmos was specifically designed for real-time systems. It is a RISC-style instruction-set architecture with features that make it time-predictable and optimized for a low WCET [16]. Patmos has an in-order, dual-issue pipeline that maximizes throughput while being time-predictable. All instructions are predicated by one of eight boolean predicate registers. If the value of the predicate is true, an instruction is *enabled*, which means it executes normally. If the predicate is false, the instruction is *disabled*. It still gets executed in the same amount of time. However, it does not read from or write to any memories or update any registers; effectively, the instruction becomes a no-op.

3.3.2 Single-Path Code

Single-path code was initially intended to make it computationally easier to perform WCET analysis. Single-path code uses predication to convert the branching control flow of a function into an instruction stream with only one execution path. To convert a function into single-path code, three techniques are used:

If-Conversion: Any conditional branching is converted into predicated instructions, such that only the needed path's instructions are enabled at runtime. The resulting code always executes all instructions in both paths, with only one path being enabled at a time. Looking at Figure 3.1, we can see the result of transforming a function to single-path code. Block `b` conditionally branches to either `c` or `d`. The color coding of Figure 3.1a's blocks matches the conditions that led to that path being taken. In Figure 3.1b, the colors indicate that only if the corresponding condition is true will the block's instructions be enabled at runtime. As such, we can see how if-conversion results in `b` always leading to first `c` and then `d`. However, only if the red condition holds at runtime will `c`'s instructions be enabled. The same holds for `d`, leading to either `e` or `f` in the traditional code, but eventually leading to both in the single-path version. Notice how we have not colored the edges in the single-path version, as they are always taken.

Loop-Conversion: Loops may iterate a variable number of times depending on runtime conditions. To avoid this variability, single-path code converts loops to always iterate the maximum possible number of times. Any superfluous iterations are instead disabled using predication. Looking at our example, we can see the function has two loops, one



(a) Traditional (b) Single-Path

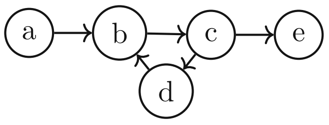
Figure 3.1: Conversion of a function with branching control flow (left) to single-path code (right).

containing the blocks `b`, `c`, `d`, and `e`, and the second containing only `f`. A single-path loop maintains a count of how many iterations have been executed and keeps looping until the maximum is reached. Inside the loop, the condition that traditionally breaks out of the loop is instead used as the predicate to all the instructions. This condition will become false at some point, meaning any further iterations will have their instructions disabled.

Function-Conversion: Single-path code also has to account for function calls. Say we have two branching paths, one of which performs a function call while the other does not. If we predicate the function call as we do for the rest of the instructions, it will not cause control to shift to the called function. This means the function's instructions are not executed (neither enabled nor disabled) if the path it was called from was disabled. Function-conversion ensures every function call is performed the same number of times, analogously to loop-conversion. Any call not logically necessary is instead disabled. Function-conversion copies all functions that are called within single-path code. The copies are then modified to take an extra predicate register argument, which specifies whether the function was called from an enabled or disabled path. The body of the copied function is then predicated on that register; if it is called from a disabled function, it will be disabled, too, and vice versa. The call instruction in the caller is not predicated, instead being provided with the predicate of the calling code to pass on. This ensures all functions are always called and executed, regardless of whether their callers are enabled or disabled.

3.3.3 Definitions

A *loop* in a CFG is a set of strongly connected nodes, i.e., a path exists between any two nodes. A *natural loop* additionally has an entry node, the *header*, which dominates all other nodes in the loop, and a *back edge* that enters the header from another node in the loop. The source node of a back edge is called a *latch*. An *exit edge* is an edge that connects a node in the loop to one outside of it. If two natural loops have the same header, we treat them as the same loop. We refer to these "merged" natural loops as a single *loop*. We define the number of iterations a loop performs as the number of times a path enters its header. All loops are either disjoint or nested within one another and



	Dom	CLDom *const*(*b*)	CLDom ¬*const*(*b*)
a	a, b, c, d, e	a, b, c, d, e	a, b, c, d, e
b	b, c, d, e	b, c, d, e	
c	c, d, e	c, d, e	
d	d	d, e	
e	e	e	e

Figure 3.2: Example CFG with the traditional dominator and constant-loop dominator relations.

identified by their headers. As such, every node has a header, which is the header of the innermost loop it is contained in. For consistency, we also consider the entire function as a pseudo-loop with the entry node as the header. Nodes without successors are *end nodes* and are assumed to return from the function.

Removing all back edges from the CFG results in an acyclic graph called a *forward control-flow graph* (FCFG). We partition the FCFG into *loop FCFGs* of the subgraphs containing only nodes whose header is the loop header. This means that for each loop, we now have a dedicated FCFG. Each node in the graph is only in one FCFG, except the headers, which reside in the FCFG of their enclosing loop and in the FCFG of the loop for which they are headers. Exit edges are represented as edges from the header of the inner loop to the original target node in the outer loop.

3.4 Constant-Loop Dominance

We consider CFGs with an optional label, `const`, on the headers of loops. If the label is present, it means paths through the loop header must visit the header a fixed number of times before exiting its loop. Otherwise, the number of visits may fluctuate between different paths. We define the constant-loop dominance relation as follows: **A node x constant-loop dominates a node y (*x cldom y*) if every path from the entry to y visits x a fixed number of non-zero times.**

Looking at Figure 3.2, we can see the traditional dominator relation (Dom) and the constant-loop dominator relations (CLDom) for the given CFG. CLDom is shown with the loop headed by `b` being both constant and variable. The most obvious difference is that when the loop is variable, none of its nodes constant-loop dominate other nodes or themselves. If the loop is constant, we can see the result is the same except `d` also constant-loop dominates `e`, unlike with traditional dominance. The last iteration of the loop must exit through `c`, meaning `d` is always visited $i - 1$ times, where i is the max iteration count. Note that constant-loop dominators behave the same as traditional dominators in the absence of loops.

Algorithm 3.1 Constant-Loop Dominators

```
    CLDom(s):                                        ▷ Starting node as input
 1: $H \leftarrow$ Inner loop headers
 2: $D \leftarrow \emptyset$                          ▷ Dominator set for nodes of $fcfg(s)$
 3: $ID \leftarrow \emptyset$                         ▷ Dominator sets for inner loops
 4: $IED \leftarrow \emptyset$                        ▷ End-Dominator sets for inner loops
 5: for $h$ in $H$ do                                 ▷ Analyze inner loops
 6:     $ID[h], IED[h] \leftarrow CLDom(h)$
 7: end for
 8: for $b \mid b \in topological\_sort(fcfg(s))$ do  ▷ Find dominators
 9:     $P \leftarrow \bigcap\{D[a] \cup IED[a] \mid \forall(a,b) \in fcfg(s) \wedge a \in H \wedge const(a)\} \cap$
            $\bigcap\{D[a] \mid \forall(a,b) \in fcfg(s) \wedge (a \notin H \vee \neg const(a))\}$
10:     $D[b] \leftarrow \begin{cases} P \cup b & \text{if } b \notin H \vee const(b) \\ P & \text{otherwise} \end{cases}$
11: end for
12: for $h, v \mid \forall h \in H \wedge \forall v \in ID[h]$ do   ▷ Extract dominators from loops
13:     $D[v] \leftarrow \begin{cases} D[h] \cup ID[h][v] & \text{if } const(h) \\ D[h] & \text{otherwise} \end{cases}$
14: end for
15: $L \leftarrow \bigcap\{header\_end\_dominators(l, D, IED) \mid \forall(l, s)\}$
16: $E \leftarrow \bigcap\{header\_end\_dominators(e, D, IED) \mid \forall(e, c) \in exit\_edges(s)\}$
17: $C \leftarrow \{c \mid \forall c \notin exits(s) \wedge \forall e \in exits(s) \wedge c \in header\_end\_dominators(e, D, IED)\}$
18: return $D, (L \cap (E \cup C)$
```

3.4.1 Algorithm

Finding traditional dominators on *acyclic directed graphs* (DAGs) is done by calculating the dominance for each node in topological order. Using topological order ensures that the dominance of a node's predecessors is established before getting their intersection to result in the dominance of the current node (and remembering to add self-dominance.)

This traditional algorithm is the basis for our algorithm for finding constant-loop dominators. It can be seen in Algorithm 3.1 on lines 8-11, where P (the intersection of predecessors) has been edited for our relation. Instead of operating on the CFG, our algorithm operates on the FCFG of the start node ($fcfg(s)$), which is a DAG. Since traditional and constant-loop dominance are equivalent when there are no loops, they are also equivalent between pairs of nodes within the same FCFG. This baseline, therefore, finds the correct constant-loop dominance between nodes in the same FCFG. The rest of the algorithm accounts for dominance between nodes of different loops (nested or in sequence).

In addition to returning the constant-loop dominator sets for each node in the given FCFG, CLDom returns a second, helper set we will call the *end dominators*. The set is calculated on lines 15-18. It represents the set of nodes in the current FCFG that would constant-loop dominate a hypothetical successor node to the FCFG's loop—assuming that node did not have any other predecessors. It is calculated by finding the nodes that constant-loop dominate all latches (L) and constant-loop dominate all exits (E) or are strictly constant-loop dominated by all exits (C). The nodes adhering to these requirements are precisely those that will always be visited a fixed number of times; they either are visited in every iteration ($L \cap E$) or will be skipped in the last iteration only ($L \cap C$). The helper function *header_end_dominators* does the following: If the given node is in the FCFG ($n \in fcfg(s)$), the function returns that node's constant-loop dominators ($D[n]$). Otherwise, the node must be in one of the inner loops. *header_end_dominators* finds the header in the FCFG ($h \in H$) whose loop contains the node; either directly or in a nested loop. If that header is constant, it returns the constant-loop dominators of the header and end dominators of that inner loop ($D[h] \cup IED[h]$). Otherwise, it returns the header's constant-loop dominators alone ($D[h]$).

Our algorithm starts by recursing on the headers of inner loops in the FCFG (lines 5-7) and storing the results for each. During dominator calculation for each FCFG node, we add the end dominators of any constant headers to their dominator sets before intersecting with the other predecessors ($D[a] \cup IED[a]$). This is what ensures that any constant-loop dominators are extracted from inner loops into the dominator sets of the current loop's nodes. Notice that *header_end_dominators* serves the same purpose in the end-dominators calculation.

Lastly, we also need to extract the constant-loop dominators of the nodes of inner loops into the current dominator set (lines 12-14). We give the loops' nodes the dominators of the header in the current dominator set, as those would not have been available in the recursive call (since $fcfg(h)$ does not contain nodes from outside the loop.) For constant loops, we also add the dominator sets of each node from their loop's recursive call ($ID[h][v]$) so they are included in the final result. Not doing so for variable loops ensures that nodes within a variable loop do not dominate anything, not even themselves.

To use CLDom for getting the constant-loop dominators of a function, we call it on the entry node and ignore the end-dominators result. It will always be empty since functions have no latches or exits.

3.4.2 Pseudo-Root Optimization

Single-path code can take advantage of constant-loop dominators to reduce execution times. The optimization focuses on those blocks that constant-loop dominate all end blocks, which means they will always be executed the same number of times per function call. We will refer to these blocks as *constant-loop dominant.*

As described in Section 3.3, function-conversion makes functions in single-path code take a predicate argument to enable or disable their bodies. However, this is not always necessary. This is most obvious for any single-path *root function*; a function that is itself single-path but is called from a non-single-path function (e.g., the main function.) A root function is guaranteed to be enabled, making the predicate argument unnecessary. The original single-path implementation recognized this and special-cased root functions not to need the predicate argument [17]. This reduces the number of instructions needed for predicate management, which results in reduced execution time.

The optimization of root functions can also be used for other functions. Any function that we can guarantee is always called enabled can be optimized as if it was a root. Taking this further, any function called from a constant-loop dominant block can also be optimized. We can do so because it means we know exactly how many times the function is called from that point, and function-conversion therefore does not need to account for variations in call numbers (as there is no variation). The callee in cases like these is called a *pseudo-root* since its code generation can be identical to a root's. Any function called from a root or pseudo-root in a constant-loop dominant block is also a pseudo-root.

The pseudo-root optimization uses constant-loop dominance to explore the call tree from the root function(s) and identifies all pseudo-root functions. The single-path transformation then uses the information to optimize all pseudo-roots to omit the predicate argument. It also changes all call instructions to pseudo-roots to be predicated, so the functions are not called when a block is disabled. Note that a function may be called from both a constant-loop dominant block and one that does not dominate. E.g., it could be called both in the entry block of a function and within only one side of a branch. In such cases, functions are duplicated, such that two versions are used: one that takes an additional argument and one that does not.

3.4.3 Implementation

We extend the open-source work presented in [17] with implementations of the constant-loop dominance algorithm and the described optimization. Patmos' compiler is based on the LLVM compiler framework [49]. Its frontend, called Clang, produces the LLVM intermediate representation called Bitcode. Bitcode is then compiled by the backend into machine code. The previous work and our extensions all reside in the backend.

We have implemented our algorithm as a `MachineFunctionPass` in the LLVM backend. At this stage, functions are in an intermediate representation close to Patmos machine code. Our algorithm is run on each function and returns a map from their blocks to the set of blocks that constant-loop dominate them. Our CFG does not have a `const` label. Instead, we provide the algorithm a function that, when given a header, returns whether it should be treated as constant. It does so by looking at the loop iteration bounds; if they are equal, the loop must be constant.

Identifying pseudo-roots is done in the `SPMark` pass of the single-path transformation [17]. We update it so that while identifying functions that will be called from a single-path context, it also identifies which calls are coming from a constant-loop dominant block and marks the target functions as pseudo-roots.

The `SPReduce` pass assigns each instruction its predicate. It also removes predication from call instructions and provides the additional predicate argument to functions. When

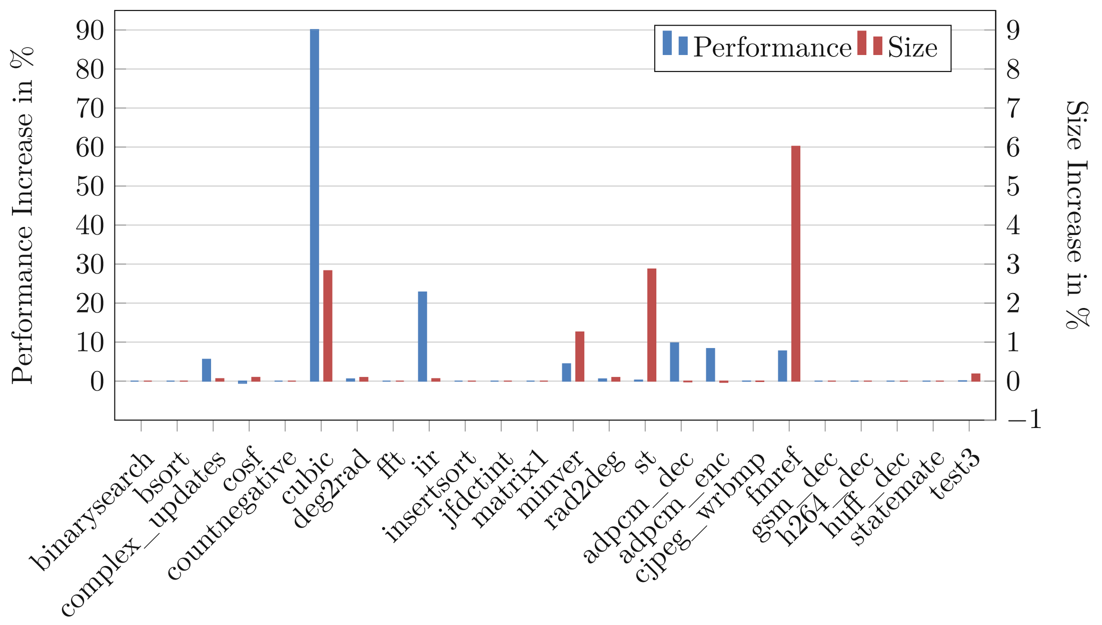


Figure 3.3: Performance and code size increase of enabling the pseudo-root optimization for single-path code.

it sees a call instruction in a constant-loop dominant block in a pseudo-root function, it omits the predicate argument, predicates the call instruction, and targets the pseudo-root version of the function (instead of the version that takes a predicate argument.)

3.5 Evaluation

We use a subset of the TACLe benchmark suite [50] to evaluate the effect of enabling the pseudo-root optimization for single-path code. We only include those programs that compile and run correctly for single-path code with and without the optimization. We exclude the `duff` program, as it has no branching and is fully inlined by the compiler, meaning no changes are made to it by the single-path transformation. The `filterbank` program is also excluded because it is so long-running that the simulator we use to run all the programs, Pasim, saturates its cycle counter, meaning we do not know what the execution times are.

3.5.1 Performance

In Figure 3.3, we show the performance increase (blue bars) of enabling the pseudo-root optimization ($\frac{disabled-enabled}{enabled} \times 100$). First, note how 11 programs see no execution time differences. All these programs—except `huff_dec` and `gsm_dec`—only have one function. This can be seen in the first row of Table 3.1, where nine programs only have one function with and without our optimization. The second row shows how many functions were recognized as pseudo-roots. For all these functions, including `huff_dec`and `gsm_dec`, only the root was recognized as a pseudo-root, meaning there is nothing to optimize.

Enabling the pseudo-root optimization produces wildly different results for the other programs. In the lower end, `cosf` sees a small performance decrease. Looking at the third row of Table 3.1, we see that many more instructions are used by single-path code with the optimization (600 $\rightarrow$ 726). The fourth row also shows an increase in the total number of call instructions (159 $\rightarrow$ 181), while the fifth row shows that there are very few calls between pseudo-roots (8). This must mean the five pseudo-root functions found did not make up for the increase in code size from duplicating three of them. On the other hand, we have the `cubic` program, which sees a 90 % performance increase. This number is all the more impressive when we look at Table 3.1. First, notice that the number of functions increases from 33 to 47. Notice also that the number of pseudo-roots found was 17 (including the root). This means that 14 pseudo-roots are also used in a non-pseudo-root context, which means two copies of each original function must be used. The rest of the functions are either only used in a pseudo-root context (3) or in a non-pseudo-root context (16). The additional copies of some functions also translate to an increased total of instructions used for managing the single-path code (627 $\rightarrow$ 921) and the number of total call instructions (136 $\rightarrow$ 215), with 58 calls being between pseudo-roots. So from where does all that performance come? The source code shows that the main function is four constant loops nested within each other. The function `cubic_solveCubic` is called four times before the loop and once in each iteration of the inner-most nested loop. Cumulatively, the main function has 879 calls to this function, all from constant-loop dominant blocks. Therefore, recognizing `cubic_solveCubic` exclusively as a pseudo-root likely produces most of this substantial increase in performance.

3.5.2 Code Size

As we have explained earlier and seen in our results so far, using the pseudo-root optimization can increase code size. The first source of this increase is the additional copies of functions used in both pseudo-root contexts and non-pseudo-root contexts. Code size can also be reduced when functions are exclusively pseudo-roots and therefore need fewer instructions for managing predicates and calling other pseudo-roots.

We measure the total size of the final executable of each program with and without the optimization and can see the result in the red bars of Figure 3.3. We can see that the difference is negligible for most of the programs that were affected by our optimization. For others, there is a significant increase but none prohibitively so. We can see no correlation between the increase in performance and code size. E.g., while `cubic` sees an enormous performance increase, it only sees a 2.8 % size increase. `fmref`, on the other hand, sees a 6 % size increase for a comparatively modest 7.7 % performance increase.

Lastly, we also need to note that the executables we have measured do not exclusively contain single-path code. They also include all original versions of any single-path

Table 3.1: Compiler statistics for each program using single-path code. For each entry, the pseudo-root optimization is disabled for the left number and enabled for the right. The metrics given are the total number of functions, the number of pseudo-root (PR) functions, the number of single-path management instructions, the total number of call instructions, and the number of calls between pseudo-roots.

	Functions		PRs		Instructions		Calls		PR-Calls	
binarysearch	1	1	1	1	19	19	0	0	0	0
bsort	1	1	1	1	32	32	0	0	0	0
complex_updates	21	23	1	10	302	264	64	64	0	21
cosf	30	33	1	6	600	726	159	181	0	8
countnegative	1	1	1	1	32	32	0	0	0	0
cubic	33	47	1	17	627	921	136	215	0	58
deg2rad	27	30	1	12	433	378	87	87	0	24
fft	1	1	1	1	91	91	0	0	0	0
iir	21	23	1	10	303	264	65	65	0	22
insertsort	1	1	1	1	34	34	0	0	0	0
jfdctint	1	1	1	1	20	20	0	0	0	0
matrix1	1	1	1	1	40	40	0	0	0	0
minver	33	36	1	4	827	922	116	143	0	9
rad2deg	27	30	1	12	433	378	87	87	0	24
st	34	48	1	18	546	854	121	200	0	46
adpcm_dec	3	3	1	3	112	96	4	4	0	4
adpcm_enc	3	3	1	3	142	126	4	4	0	4
cjpeg_wrbmp	3	3	1	3	71	64	4	4	0	4
fmref	72	103	1	36	1458	2070	377	523	0	78
gsm_dec	3	3	1	1	389	389	8	8	0	0
h264_dec	1	1	1	1	116	116	0	0	0	0
huff_dec	2	2	1	1	199	199	5	5	0	0
statemate	1	1	1	1	47	47	0	0	0	0
test3	101	101	1	101	1510	1210	200	200	0	200

function, any initialization code that eventually calls the benchmark function, and the standard library. This means the size differences are likely bigger for both increases and decreases in a real-world, single-path-only scenario.

3.5.3 Source Access

Patmos and its platform, T-CREST [15], are available as open-source and include the contributions of this paper. The Patmos homepage can be found at http://patmos.compute.dtu.dk/ and provides a link to the Patmos Reference Handbook [51], which includes build instructions.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-llvm-project (commit hash: 82eb73bff7336674027afecb254f1e3ebd1c23c2).

3.6 Conclusion

In this paper, we presented the constant-loop dominance relation and how it can be used for optimizing single-path code. We first defined the relation as a variation of the traditional dominance where the number of visits to a node must be constant. This takes loop bounds into account to recognize constant loops. We then presented a recursive algorithm for finding the constant-loop dominators. It first explores (nested) loops and uses the intermediate results for the outer loops. We showed how the relation can be used to identify pseudo-root functions in single-path code. These have the quality of being called a fixed number of times. We used this property to optimize single-path code to require fewer instructions to manage predicates and to reduce unnecessary calls. Our evaluation showed sporadic but significant performance improvements from applying our optimization. While some programs saw no execution-time differences, others saw an up to 90 % performance increase. We also showed that the optimizations do affect code size, with executable sizes increasing by up to 6 %.

CHAPTER 4

Towards Dual-Issue Single-Path Code

Abstract—*The Patmos instruction-set architecture is designed for real-time systems. As such, it has features that increase the predictability of code running on it. One important feature is its dual-issue pipeline: instructions may be organized in bundles of two that are issued and executed in parallel. This increases the throughput of the processor in a predictable manner, but only if the compiler makes use of it.*
Single-path code is a code-generation technique that produces predictable executions by always following the same trace of instructions. The Patmos compiler can already produce single-path code, but it does not use the second issue slot available in the processor. This is less than ideal because the single-path transformation results in code that has a high degree of instruction-level parallelism.
In this paper, we present a single-path code generator that can produce bundled instructions. It includes generic support for bundling algorithms, such that implementing them is simple and does not require changing other parts of the compiler. We also present one such bundling algorithm plugged into the single-path code generator. With it, we show that we can produce dual-issue instructions to improve performance.

4.1 Introduction

Patmos [16] has been designed to support the single-path programming paradigm [13]. The following two features have been added for the support: (1) predication of each instruction and (2) a dual-issue pipeline. Predicates are needed for disabling the execution of the *not-taken* part of a decision; when both paths are executed as is the case in the single-path paradigm. The dual-issue pipeline may improve the performance of single-path code by running instructions in parallel.

Predicates are used by the Patmos compiler when generating single-path code [17]. However, the dual-issue pipeline of Patmos is not explored when generating single-path code. For conventional code, the current instruction scheduler for the Patmos compiler is only able to improve performance by about 11 % when using the dual-issue pipeline [16].

In single-path code, both branches of an if/else condition are executed. However, their instructions will be predicated with the condition, such that the predicate will only be true for the branch that is meant to be taken. A branch whose predicate is true will execute as usual. For the other branch, the predicate is false and its instructions will not produce any side-effects, i.e., no results are written back to the register file or main memory. Therefore, a false branch will effectively contain only *no-ops*, even though all instructions are executed. Since only one of the two alternatives affects the state of registers or memory, they have no data dependency. This means we can execute them in parallel with the dual-issue pipeline of Patmos.

This paper presents an instruction scheduler for single-path code that uses the two issue slots of the Patmos pipeline. Where previous work produced single-path code that uses only the first issue slot [17], this paper presents a technique for transforming the code such that the second slot is used, too. The contributions of this paper are: (1) a description of a generator for dual-issue single-path code that can support any instruction bundling algorithm, (2) an implementation of an automatic dual-issue single-path code generator in the Patmos compiler, and (3) an evaluation of the performance gain achieved over single-issue single-path code.

The rest of the paper is organized in 5 sections: The following section presents related work. Section 4.3 provides background on the Patmos processor and single-path code generation in general. Section 4.4 details how code is transformed into single-path form and how bundling is supported. Section 4.5 describes the implementation in the Patmos compiler together with presenting a bundling algorithm. Section 4.6 evaluates the performance impact of the dual-issue code compared to single-issue code. Section 4.7 concludes the paper.

4.2 Related Work

The single-path code generation approach is introduced in [70] for use in real-time systems. The authors of [17] continue the work, presenting an algorithm for generating single-path code from conventional code. They show that the generated code can be used with the Patmos architecture, but also that the performance cost of the conversion is significant but manageable. We build on this work, such that the single-path code generated now makes use of the second issue slot of the Patmos architecture.

Similarly to this paper, the authors of [71] investigate how to make use of very-long instruction word (VLIW) architectures for time-predictability. Even though their work is based on single-path, they limit their practical investigation to basic blocks of innermost loops and do not use loop transformations nor support inter-procedural code. For bundling, they use hyper-block scheduling introduced in [72].

The authors of [73] build on both works, presenting a variation of the hyper-block formation algorithm that takes worst-case execution time (WCET) into account to better select which blocks to merge into hyper blocks.

In [74], the authors investigate the problem of constructing hyper blocks such that WCET is minimized on clustered VLIW processors—a type of processor with many functional units, perfect for code with high instruction-level parallelism.

In [75], the authors present a memory hierarchy specifically tailored to make use of the properties of single-path code to significantly improve performance without impacting predictability. They use a prefetcher that exploits single-path code to reduce instruction-cache miss rate and its penalty. The effects of caches on single-path code are also addressed in [76]. The authors present a technique for aligning single-path loops with the instruction cache to reduce cache misses during loop execution.

Both [77] and [78] investigate how to predictably execute code on traditional architectures. The former investigates the impact of adding single-path code support to an existing architecture by introducing instructions like conditional moves. They show that this can be a worthwhile effort, depending on the coding style used and the specific application. The latter also investigates different code generation techniques to make execution more predictable. They use software techniques to eliminate timing anomalies originating from the processor's out-of-order pipeline and to control the state of the dynamic branch predictor.

Lastly, in [79] the authors address two issues with single-path code that we do not address in this paper: (1) it requires special architectural support (like conditional moves) and (2) it increases power consumption since more code is run. They present techniques to address both issues at the cost of a slight reduction in predictability and increased execution time but achieve an increase in power efficiency. In contrast, our work will increase the utilization of the Patmos processing core and can, therefore, be expected to increase power consumption.

4.3 Background

In this paper, we build on two technologies: the Patmos processor and single-path code. The following subsection describes the time-predictable processor Patmos, which has been designed to efficiently execute single-path code. It is followed by a description of the principles of single-path code and a set of definitions used in the rest of the paper.

4.3.1 The Patmos Processor

Patmos is a RISC style processor optimized for real-time systems [16]. The aim of Patmos, and the whole T-CREST architecture [15], is to build time-predictable computers [80]. It uses an in-order pipeline to avoid any timing anomalies [81]. As the analysis of caches may introduce large over-estimations of the WCET, Patmos contains special caches and scratchpad memories to reduce these estimates. Instructions are cached in the so-called method cache [82, 45]. It caches full functions, such that cache misses can only occur at function-call boundaries. Patmos also contains a stack cache for stack-allocated data [46].

To support single-path code, all instructions in Patmos can be predicated with one of the eight predicate registers (or their negations). These predicates are used in single-path code to remove execution variation, e.g., by using branchless conditional execution. Additionally, instructions have the same timing regardless of the value of their predicate; a multiplication instruction with a predicate set to false is not faster than one with the predicate set to true. Only updating the processor state, i.e., write-back into the register file or a write into the memory, are affected by the value of the predicate. We say an instruction is *enabled* if its predicate evaluates to true when executed. If the predicate evaluates to false, we say the instruction is *disabled*.

Since single-path code executes many data-independent instructions, Patmos is a dual-issue architecture, enabling the execution of two such instructions in parallel. They need to be scheduled by the compiler and marked as a dual-issue instruction pair, called a *bundle*. The marking of a bundle is a single bit in the first instruction and can be decoded in the fetch stage. This stage uses a split cache (for even and odd addresses) and always fetches two instructions. If the instruction is marked as a bundle, both instructions are used and the program counter is advanced by two instructions. If not, only the first instruction is used and the program counter is advanced by one instruction.

4.3.2 Single-Path Code Generation

We call a piece of code *single-path code* if its execution enforces the same unique instruction trace, i.e., the same sequence of instructions and accesses to instruction memory for all possible data valuations of the variables that are manipulated. The point of single-path code is that the enforcement of an invariable sequence of accesses to instruction memory eliminates one of the central sources of timing unpredictability. In particular, when executing single-path code multiple times from the same processor and memory system states (i.e., the same state of the processor pipeline and instruction cache) and when the execution times of instructions are constant, the execution time of the entire code can be expected to be constant.

Constant execution time makes timing repeatable, which brings along the following desirable properties:

- It allows for the most precise argumentation about code timing. In the simplest cases—where there is no timing variation from the memory hierarchy—the code will always have the exact same timing. This makes WCET analysis as simple as running and measuring the code and produces an exact result.
- When execution times are expected to be invariable, any deviations from the expected timing can be taken as error indicators. Thus, monitoring the execution time of single-path code is a simple but powerful error-detection mechanism.
- An observation of execution times does not provide any clues about the performed computations. This means that single-path code safeguards computer systems against side-channel attacks that use execution monitoring to get hints about what is happening in the code. This contributes to computer systems security.

The fact that we use single-path code may seem to limit the applicability of the presented approach to algorithms that do not contain any data-dependent control decisions. Such a limitation is, however, not the case. Single-path code is generated by a compiler that applies special code transformations to eliminate data-dependent control flow from the input source code. Thus, any execution-time-bounded code may be used as a source for single-path code generation. Hard real-time code has to be execution-time bounded, which means the maximum number of loop iterations and calls to recursive functions must be bounded [83].

We use three transformation techniques to create single-path code:

If-conversion

When executing branches, timing variability can be introduced when the two possible paths have different lengths. To address that, we instead predicate the two alternatives on the value of the branch condition and then make both paths execute. Thus, if the condition is true, only the true-path code's predicate is set to true, and vice versa for the other path. The effect is that we effectively only run the required path, but the timing is constant, as both paths' code is executed (with the false path being disabled and therefore having no effect.)

Figure 4.1 illustrates if-conversion. On the left, we see conventional code that will always branch over one of the execution alternatives. On the right, predicated execution never branches, but will instead always disable one of the alternatives.

Loop-conversion

Loops are another source of timing variability. If the number of iterations taken by the loop changes, the time it takes to execute the loop—and therefore also the program—

```
if !cond goto Lelse
Lthen:
x = a + 1                  ( cond) x = a + 1
goto Lend                  (!cond) x = b - 2
Lelse:                                .
x = b - 2                             .
Lend:                                 .
...                                   .
```

Figure 4.1: The difference between branching and predicated execution. On the left, a condition makes the execution skip one of the paths, while on the right both paths are executed, but only one of them will be enabled at a time.

changes, too. To eliminate this variability, we transform the loop, such that it will always iterate the maximum number of times. However, to maintain the semantics of the program, we use predication to disable the loop body as soon as the required number of iterations have been executed. Thus, any superfluous iterations have no effect but are still run to maintain constant timing.

Procedure-conversion

A final source of timing variability comes from the calling and execution of procedures. Even if the execution of a procedure takes constant time, if that procedure is called a variable number of times, then the program's timing will also be variable, e.g. if one path of a branch calls the procedure, but the other doesn't. To maintain constant timing, procedures are called and executed even though the calling code is disabled (e.g., from a disabled path in a branch.) However, all procedures accept an additional predicate argument, which is used to predicate the execution of the entire procedure. If the call stems from disabled code, then the procedure body is also disabled. This conversion ensures that all procedures are always called a fixed number of times. Since each call has constant timing, the whole program will also have a constant execution time.

4.3.3 Definitions

Basic Block (BB): A sequence of instructions whose execution always starts at the first instruction and may only branch on the last.

Control-Flow Graph (CFG): A directed graph of BBs where the edges model how control flows from one block to the next. A branch is modeled as a block that has two out edges in the CFG—one for each path. We do not handle branches with more than two targets. As such, `switch`-like behavior must be converted into a cascade of simple alternatives.

Dominate: A block dominates another block if all paths leading to the latter must first go through the former.

Post-dominate: A block post-dominates another block if all paths going through the latter must eventually go through the former, too.

Loop Header: In a loop, the header block is the one that dominates all the other blocks in the loop, i.e., it is the entry to the loop. We associate every block in the CFG with the header of the inner-most loop containing it. We also treat the whole procedure as a pseudo loop, where the initial block of the procedure is a header too. Therefore, all blocks in the procedure have a header (except the procedure's initial block.)

Back Edge: An edge whose source block is in a loop and the target block is the header of the same loop.

Exit Edge: Has the source block in the loop but target block outside it. Informally, the edge exits the loop.

Forward CFG (FCFG): An acyclic CFG that is the result of removing all back edges from a CFG.

Control Dependence: In a CFG—given the blocks `x`, `y`, and `z`—`x` is control dependent on `y` if `x` post-dominates `z` but not `y`. We also say that `x` is control dependent on the edge `(y,z)`.

Equivalence Class: Two blocks are in the same equivalence class if they are control dependent on the same set of edges.

Guard: A predicate or register guards an instruction if its value determines whether the instruction is enabled or disabled. A block is guarded by a predicate or register if any of its instructions are guarded by the same.

4.4 Single-Path Transformation

The single-path code generation technique takes the CFG of a procedure and rearranges it—with various edits—to produce straight-line code. Our transformation is a variation of the one presented in [17]. In this section we will describe the whole transformation informally, highlighting the differences from the original work.

4.4.1 Preparing the CFG

When transforming conventional code into single-path, we start by analyzing the CFG of the procedure in a similar way to how the original single-path transformation would: We identify all loops by their header blocks and find which other blocks are contained in each loop. For each sub-CFG in the procedure, we construct an FCFG: We create 2 new nodes in the graph, `s` and `t`. We connect `s` to `t` and to the header. Then, for each back or exit edge, we connect the source block to `t`.

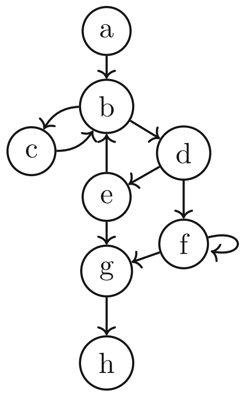


Figure 4.2: A control-flow graph of a procedure.

We use the procedure in Figure 4.2 as an example. From it, the FCFG generated from the loop with header `b` can be seen in Figure 4.3. In Figure 4.4 we can see the FCFG of the pseudo loop with header `a`. In it, we see that nested loops are only represented by their headers; the blocks `b` and `f` are the headers of the two loops in the procedure.

We use FCFGs to partition the graph into equivalence classes. Looking at Figure 4.4, we have two equivalence classes: {`a`,`b`,`g`,`h`} and {`f`}. Each class is assigned a unique predicate that will become the guard for its instructions.[1]

The original transformation tracked guard predicates on a per-block basis, as all instructions in a block would, in the end, be guarded by the same predicate. However, for our work, we will end up bundling blocks with different predicates. This means the resulting blocks will have some of their instructions guarded by one predicate and the others by another. Therefore, we must track predicates on a per-instruction basis, such that when blocks are bundled, the instructions maintain the predicate befitting the equivalence class they are part of.

To disable the header block whenever an exit edge is taken, we add any non-exit edges, that are outgoing from the source block of any exit edge, as control-dependence edges of the header. By having these edges assign the header's predicate to false, we ensure that superfluous iterations disable the whole loop body.

At this point in the original single-path transformation, code generation would begin by reordering the CFG into a straight-line sequence. However, our approach introduces a preceding step for the bundling of blocks.

4.4.2 Bundling

At this point in the transformation, we have enough information about the (F)CFG to start block bundling. When bundling two blocks, we issue the first block's instructions in the first issue slot of the Patmos pipeline and the second block's instructions in the

[1]This is an abstract predicate that is later assigned a physical predicate register.

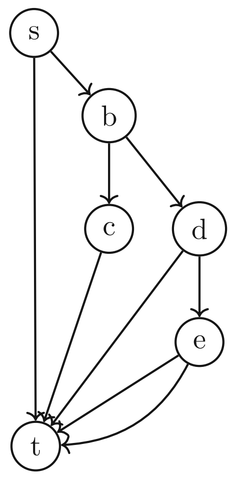


Figure 4.3: FCFG of the loop b.

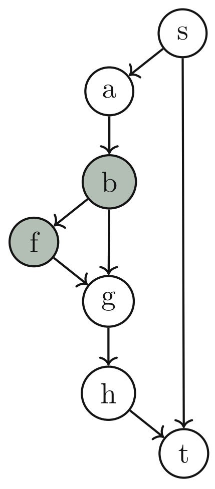


Figure 4.4: FCFG of the pseudo loop a, with the headers of loops b and f in grey.

second issue slot.[2] An exact algorithm for which block pairs to bundle will be given in the next section. In this subsection, we will describe how an algorithm can fit into this single-path transformation.

For any two blocks to be bundleable into a combined block, three rules must hold:

- **Non-equivalence:** The pair cannot have the same equivalence class since that would mean that their instructions will be enabled simultaneously. We cannot allow this, as that would mean the two blocks might interfere with each other, e.g., by using the same registers.

[2]Not all Patmos instruction can be issued in the second issue slot, e.g., memory loads and stores can only be issued in the first slot.

- **Non-domination:** Neither block can dominate or post-dominate the other. This ensures that the semantics of the program are maintained, as a pair of blocks can be in different equivalence classes but also have one dominate the other. For example, one can be a branching block and the other can be a block from one of the conditional paths. If they were to be bundled, then the code for calculating the branch condition would be executed at the same time as the code for one of the paths—which might not be taken.

- **Looping:** They must be in the same loop. This is to ensure that each block is iterated over the correct number of times.

Any bundling algorithm must ensure that any block pairs it bundles adhere to the above rules. It must consist of two parts: (1) `findBlockPair`: Finds a pair of blocks to bundle and (2) `bundleBlockPair`: Does the actual bundling of a pair of blocks, deciding exactly which instructions in each block get bundled with each other.

The algorithm is called continuously by the transformation procedure, first calling `findBlockPair` and then `bundleBlockPair` on the returned pair. This setup is flexible enough to support a wide range of algorithms. Greedy algorithms, that do not store the state between finding block pairs, are most naturally implemented in this setup. However, it also allows for the storage of state between calls, for algorithms that need that. The setup should, therefore, be able to support all types of algorithms.

After each call to `findBlockPair` and `bundleBlockPair`, the (F)CFG is automatically reordered such that the semantics of the program are maintained: edges to or from the original blocks are moved to the new bundled block, such that control flow is maintained. The same is done for predicate definitions, which must maintain the predicates they were originally guarded with.

The bundling algorithm is continuously called until it can no longer find any more pairs to bundle, at which point we can continue to the final code generation. However, it is worth noting, at this point, that the algorithm is free to reuse an already bundled block to bundle with another block, meaning more than two of the original blocks are then inside the resulting block. Even though the Patmos pipeline can only issue two instructions at a time, this single-path transformation allows for any number of unbundled blocks to be bundled into one mega-block, as long as all block-parings adhere to the rules.

4.4.3 Code Generation

The final code generation is done in much the same way as the original single-path transformation. For each edge in the FCFG that has an equivalence class depend on it, we insert predicate definitions at its source block. This is code that calculates the class's predicate from the condition that leads to the edge being taken. A key difference from previous work is that multiple equivalence classes can depend on a given edge; its target block could be the result of bundling.

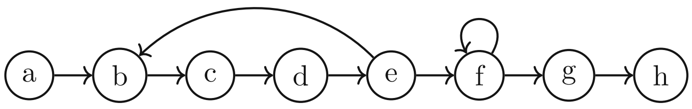


Figure 4.5: The single-path CFG transformed from the one in Figure 4.2. No bundling is performed.

We can now begin reordering the blocks into a straight-line sequence based on the FCFG. All blocks in the loop are put into a sequence using topological order—excluding `s` and `t`, which are no longer needed. Each block is given an edge to the next block in the sequence, eliminating any conditional control flow. A branch is inserted from the last node to the header (which is always the first in a topological ordering.) The branch is made conditional on the number of times it is taken: After N-1 times (with N being the maximum number of iterations) the condition becomes false and the loop is exited. This ensures we always run the loop the maximum number of times.

In an outer loop's FCFG we only see the headers of nested loops. Instead of connecting these headers to the next block in the sequence, their own FCFG's final node is connected to the next node in the outer FCFG. In Figure 4.5 we can see the result of the transformation of the CFG in Figure 4.2. We can see how the FCFG of the loop with header `b` ends with the block `e`, which is connected to the next block in the outer loop, `f`.

4.5 Implementation

We build directly on the open-source implementation presented in [17]. We adapt it with the support for bundled blocks described in the previous section and add a dedicated pass that takes single-path functions and bundles their blocks.

4.5.1 Compiler Overview

The Patmos compiler is based on the LLVM compiler framework [84]. LLVM provides a frontend called Clang, which translates C source files into an intermediate representation called Bitcode. This representation is very low level, being similar to assembly code but not specific to any hardware architecture. Bitcode is therefore well suited as a target for generic language- and target-independent optimizations which are provided by the framework. We link together the user's code, standard library, and support libraries in Bitcode. This gives the backend implementation a whole-program view for analysis and transformations. This is important for the single-path transformation since it can affect all code—not just the user's code. The backend takes the Bitcode and translates it into machine code for the Patmos architecture.

4.5.2 The Single-Path Passes

Unchanged from the original, the single-path transformation starts with a set of passes that prepare the code for being transformed into single-path. This includes passes that ensure certain properties are established and a pass that copies code that needs to be present in both single-path and conventional versions. The rest of this section only concerns the single-path versions.

Single-Path Info Pass

A pass dedicated to analyzing the CFG of each function; finding loops, building FCFGs, and assigning equivalence classes and predicate definitions to each block. The result of this pass is used, and edited, by the next passes to correctly bundle and emit the code.

We change two properties in this pass compared to the original implementation. First, we assign equivalence classes to each instruction instead of each block, which we have argued for before. We also track predicate definitions from this point. In the original implementation, this was not needed, as the information was gathered in the Reduce pass described later. However, block bundling corrupts the information about which guards must be put on each definition. Therefore, we gather this information now and maintain it throughout block bundling to be used in the Reduce pass.

Bundling Pass

A dedicated pass implements the bundling algorithm. As a proof-of-concept, we implement a very simple bundling algorithm:

`findBlockPair`: For each loop, we go through all the blocks, looking for any that have exactly two immediate successors. When we find one, we ensure the following about its successors:

- The successor edges are neither back nor exit edges.
- Neither of them is a header of a nested loop.
- Neither of them post-dominates the other.

If these three requirements hold, the two successors are a pair of bundleable blocks. We repeat this until we cannot find a block with successors to merge. We do this for all loops, starting at the header and following a depth-first traversal.

`bundleBlockPair`: We simply try and bundle the first instruction in the first block with its counterpart in the second block. This is not always possible, as some instructions in the Patmos architecture can only be issued in the first issue slot—loads and stores from memory have this property. Alternatively, we try to switch them, such that the second block's instruction is in the first issue slot. If this is not possible either (e.g. if

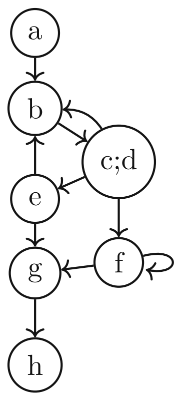


Figure 4.6: CFG where c and d have been bundled.

both instructions are loads,) we simply revert to interleaving them without bundling. We then do the same for the next pair of instructions. If one block has more instructions than the other, we append the additional instructions to the end.

We can show that any pair found by `findBlockPair` is bundleable:

- **Non-equivalence:** If the block whose successors we are bundling—the parent block—is not itself a result of bundling, then it is trivial to see that the successors must be in two different equivalence classes. If the parent is a bundled block, we have two possibilities: (1) Each candidate is the successor of one of the original parent blocks. We know that the two parent blocks are of different equivalence classes, which means their successors cannot be of the same equivalence class. (2) The candidates are the successors of the same parent, which we have already argued for.
- **Non-domination:** We specifically check any candidate pair for whether this rule holds and, if not, disregard them.
- **Looping:** Since we disallow block pairs that have exit edges from the parent, we know they cannot be in an outer loop of the parent block's loop. Since we also disallow them being headers of inner loops, the only possibility left is that they are both part of the parent block's loop.

Using this algorithm on the CFG in Figure 4.2 will result in the bundling of the blocks c and d as can be seen in Figure 4.6.

Reduce Pass

The reduce pass produces the final single-path code. First, a specialized predicate-register allocation is performed. It assigns predicate registers to each equivalence class. If not enough registers are available, a general-purpose register is used as a spill location, such

Table 4.1: Execution time (in cycles) for different TACLe benchmark programs with or without bundling enabled. Omitted the 5 programs that produced no bundles.

	Original	Bundled	Difference	Improvement %
synth_opt	586 778	316 442	270 336	46,07
bsort	428 418	428 218	200	0,047
countnegative	43 321	42 439	882	2,036
prime	24 893	24 844	49	0,197
adpcm_dec	7 173 386	7 173 380	6	0

that we can spill the predicate register that is used furthest in the future, freeing it for reuse.

This register-allocation algorithm is the same as the original work's algorithm, except modified to be able to handle bundled blocks, which is not much different from handling non-bundled blocks with just one guard predicate.

After predicate-register allocation, we go through all the instructions in the function and assign them the physical predicate-register guard that the allocation specifies. Function calls are never predicated since the functions need to be run every time. Instructions for spilling or restoring are inserted in newly created BBs and are also not predicated.

Finally, after reordering the blocks into a straight-line sequence, consecutive ones are merged wherever possible, such that the remaining branches facilitate looping.

4.6 Evaluation

We implemented a simple bundling algorithm to prove the effectiveness of the single-path transformation in supporting any bundling algorithm. To show that it works and increases performance, we measure the number of cycles used when executing different benchmark programs with and without bundling. The benchmarks are run on a cycle-accurate simulator of the Patmos processor, where all caches are 2-way set associative using a least-recently-used replacement strategy.

4.6.1 Synthetic Benchmarks

Theoretically, a 50 % improvement in execution time is the maximum possible when using bundling on the Patmos processor. This occurs when all instructions are paired up in bundles, cutting the number of instructions in half. However, it is practically impossible to reach this ceiling, as there are various reasons that inhibit some instruction from being bundled, e.g., arithmetic instructions with immediate values larger than 4095 take up two instruction words on their own, which means no other instructions can be bundled with them.

To get a feel for the potential of bundling instructions, we devise two synthetic benchmark programs that are easy for our naive algorithm to bundle. With the first benchmark, `synth_opt`, is designed to maximize the performance improvement our algorithm can achieve. As the algorithm always only bundles the two branches of an if/else statement, this benchmark puts some heavy math calculations in each branch of such a statement. We also ensure that the two branches do the exact same amount of math, such that all their instruction can be bundled, with no spare instructions having to forego being bundled. Lastly, we put the if/else statement inside a loop that iterates 4095 times. This number of iterations is the maximum possible, were the loop counter can still fit into short-immediate arithmetic operations. This is required, as our math works on the loop-counter to ensure it cannot be optimized away. Any immediate values larger than 4095 are so large the compiler has to put them in a long-immediate arithmetic instruction which, as stated before, takes up two instruction words and cannot be bundled with other instructions. The result is a program that is primarily a loop with 142 instructions. When bundling is enabled, 132 of the instructions are bundled by our algorithm. The last 10 are from the overhead of managing the loop iteration and cannot be bundled by our algorithm.

Table 4.1 shows the number of clock cycles used for the execution of each benchmark program without bundling (*Original*) and with bundling (*Bundled*). It also show the difference between the two counts and the resulting improvement in execution time.

For our `synth_opt` benchmark, we can see that without bundling the program requires 586 778 cycles to execute, while with bundling it only requires 316 442 cycles. This is an improvement in the execution time by roughly 46 %. This almost exactly matches the reduction in the number of bundles in the loop. Since 132 instructions were bundled together, 66 cycles were removed from the original 142 cycles required to run the loop, which is roughly a reduction of 46,5 %. The last half percentage point not reflected in our results can be attributed to the instructions run before the loop even starts.

4.6.2 TACLe Benchmarks

we measure the number of cycles used by 9 programs from the TACLe benchmark [50], with and without bundling. Our compiler currently fails to generate single-path code for the rest of the benchmark's programs, which means it is also unable to bundle them.

However, only 4 of the 9 programs measured resulted in bundles being created when bundling was enabled. These are therefore the only programs shown in the table. This is caused by the simplicity of the bundling algorithm, which is not sophisticated enough to see bundling opportunities in the 5 remaining programs.

To get a more detailed look at what the algorithm is doing, we can look at Table 4.2. It shows compiler statistics regarding the bundling of blocks for each of the 9 programs. First are the number of basic blocks in the program (*Before*), then the number of pairs of blocks the algorithm found suitable for bundling (*Pairs Bundled*), and lastly the pairs it did not find suitable (*Pairs Rejected*). By "pairs" we mean two blocks that have the same parent and are then checked for whether they can be bundled. We can see that

Table 4.2: Statistics on the number of blocks in each compiled program from the TACLe benchmark.

# Blocks	Before	Pairs Bundled	Pairs Rejected
bsort	17	1	7
countnegative	15	1	5
prime	22	2	11
adpcm_dec	60	2	21
insertsort	20	0	8
jfdctint	14	0	4
matrix1	22	0	7
md5	55	0	17
petrinet	163	0	128

Table 4.3: Statistics on the number of instructions in each compiled program from the TACLe benchmark. Omitted the 5 programs that produced no bundles.

# Instructions	Before	Pairs Bundled	Pairs Interleaved
bsort	147	3	0
countnegative	159	2	0
prime	245	2	0
adpcm_dec	1261	2	0

at most two pairs of blocks were ever bundled, and for the majority of programs, many candidates were rejected.

Looking at the four programs that had blocks bundled, we take a look at whether `bundleBlockPair` found the optimal instruction bundling. A sub-optimal bundling would revert to interleaving instructions. Table 4.3 shows the total number of instructions in the programs (*Before*), then the number of instruction pairs that were bundled (*Pairs Bundled*), and lastly the number of instruction pairs that should have been bundled but could not because they both needed to be in the first issue slot (*Pairs Interleaved*). We can see that all instructions that could have been bundled were bundled. Therefore, a better algorithm for `bundleBlockPair` would have made no difference.

In general, we can see that bundling does in fact increase performance, though, as we expected, not by much. For most of the programs that had their blocks bundled, the increase is negligible, except for the *countnegative* which exceeds our expectations at a 2 % performance increase. This is impressive when we note that only 2 bundles were created out of the 159 instructions.

Source Access

Patmos and the T-CREST platform are available as open-source and include the contributions of this paper. The Patmos homepage can be found at http://patmos.compute.dtu.dk/

and provides a link to the Patmos Reference Handbook [51], which includes build-instructions in Section 5.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-llvm.[3] However, we advise following the handbook instructions to correctly set up a machine to build and run Patmos programs.

4.7 Conclusion

In this paper, we presented a generator for single-path code that can use the second issue slot of the Patmos architecture's dual-issue pipeline. We present an implementation of the generator for the compiler of the Patmos processor. We show that the transformation can incorporate a wide range of bundling algorithms, and also present a simple one, that takes single-issue single-path code and produces dual-issue single-path code.

We evaluate the work on a subset of the TACLe benchmark suite and show that dual-issue code is produced and improves performance. Even though the increase is small, we show that the impact of using the second issue slot of the Patmos architecture has the potential for performance gains. More work is therefore needed to more effectively bundle instructions, such that the potential of pairing single-path code with the Patmos architecture can be better realized.

[3]The evaluations were done using the commit with hash: 3e88062e4e1fa5ec7a4ffcfb4b865a913914bc65.

CHAPTER 5

Compiling for Time-Predictability with Dual-Issue Single-Path Code

Abstract—*Designed for real-time systems, the Patmos instruction-set architecture's features ensure a high degree of predictability. One such feature is its dual-issue pipeline, which can issue and execute bundles of up to two instructions at a time. Executing instructions in the second issue slot is a predictable way to increase the throughput of a processor, but without dedicated support from the compiler, this benefit cannot be unlocked. A compiler generates highly predictable programs by generating single-path code. This technique produces code that always follows the same trace of instructions. While Patmos' compiler can already produce single-path code, it does not assign any instructions to the second issue-slot. This limitation is unfortunate, as single-path code inherently possesses a high degree of instruction-level parallelism. In this paper, we present a single-path code generation technique with support for dual-issue pipelines. It can also support different bundling algorithms, which allows changing algorithms without having to edit other parts of the compiler. We present a simple bundling algorithm plugged into the single-path code generator. It looks for branches and bundles the basic blocks on each path of the branch. While this specific bundling algorithm is too simple to provide a real-world benefit, it highlights the potential that further work on bundling algorithms can unlock.*

5.1 Introduction

Single-path code allows time-predictable execution of real-time applications [13]. It allows for the most precise argumentation about code timing. When there is no timing variation from the memory hierarchy, the application will have a constant execution time. This makes worst-case execution time (WCET) analysis as simple as running the code and measuring the execution time. To enable single-path code execution, a processor and the memory hierarchy need to fulfill several properties, e.g., have a constant execution time of instructions and predicated instructions.

Patmos [16], as part of the T-CREST project [15], has been specially designed to support the execution of single-path code: (1) each instruction can be predicated and (2) a dual-issue pipeline can execute code from both branches of an *if-converted if/else* decision. Then, predicates disable the execution of the *not-taken* part of a decision when both paths are executed, as in the single-path paradigm.

Patmos' compiler uses predicates when generating single-path code [17]. However, the dual-issue pipeline of Patmos is not exploited when generating single-path code. The current instruction scheduler for the compiler can only improve performance by about 11 % when using the dual-issue pipeline [16] on traditional code.

In single-path code, both branches of an *if/else* condition are executed. However, only one of the two branches will have a predicate set to `true`. The other branch, executed with its predicate set to `false`, will behave like a sequence of *no-ops*, even though all instructions are executed. This mutually exclusive predicate activation of alternative branches renders any (seeming) dependencies between instructions in the alternative branches ineffective, i.e., the sharing of register names or addresses by these branches creates only so-called *false* dependencies and does not create any resource conflicts at runtime. A compiler generating single-path code knows about these false dependencies. Therefore, it can use two alternative branches' instructions as ideal candidates for parallel issue in single-path code generation.

This paper presents an instruction scheduler for single-path code that uses the two issue-slots of the Patmos pipeline. Previous work produced single-path code that uses only the first issue slot [17]; this paper presents a technique for transforming the code such that the second slot is used as well.

The contributions of this paper are: (1) a description of a generator for dual-issue single-path code that can support any instruction bundling algorithm, (2) an implementation of an automatic dual-issue single-path code generator in the Patmos' compiler that uses a simple proof-of-concept bundling algorithm and (3) an evaluation of the performance gain achieved over single-issue single-path code. This paper is an extension of [85]. In this paper, we make a slight change to the specific bundling algorithm used as a proof-of-concept to handle more complex cases, e.g., nested if/else statements. We also expand our evaluation to include a set of synthetic benchmarks to give a better feel for the potential of using the second issue slot. Additionally, we use a larger subset of the

TACLe benchmark and a real-world drone control program. In the rest of the paper, when we mention the *original work*, we refer to the work presented in [17].

The paper is organized in 7 sections: The following section presents related work. Section 5.3 provides background on the Patmos processor and single-path code generation in general. Section 5.4 details how code is transformed into single-path form and how bundling is supported. Section 5.5 describes the implementation in the compiler together with presenting a bundling algorithm. Section 5.6 evaluates the performance impact of the dual-issue code compared to single-issue code. Section 5.7 concludes the paper.

5.2 Related Work

The single-path code generation approach is introduced in [70] for use in real-time systems. The authors of [17] continue the work, presenting an algorithm for generating single-path code from conventional code. They show that the generated code can be used with the Patmos architecture and that the conversion's performance cost is significant but manageable. We build on this work such that the single-path code generated now uses both issue slots of the Patmos architecture.

Single-path code is dependent on architectural support—like conditional moves, at a minimum—to work. This precludes its use on existing architectures without this support. To alleviate this, the authors of [86] present a filter that can be added to existing processor cores and alters the instruction stream to produce the same effect as single-path code. The filter monitors the state of the processor and the control flow of the program. It then generates instructions for the processor that are effectively single-path. This is done by substituting any non-taken code paths with `no-op` instructions.

Both [77] and [78] investigate how to execute code on traditional architectures predictably. The former investigates the impact of adding single-path code support to an existing architecture by introducing instructions like conditional moves. They show that this can be a worthwhile effort, depending on the coding style used and the specific application. The latter also investigates different code generation techniques to make execution more predictable. They use software techniques to eliminate timing anomalies originating from the processor's out-of-order pipeline and control the dynamic branch predictor state.

In [75], the authors present a memory hierarchy specifically tailored to use single-path code's properties to improve performance significantly without impacting predictability. They use a prefetcher that exploits single-path code to reduce instruction-cache miss rate and its penalty. The effects of caches on single-path code are also addressed in [76]. The authors present a technique for aligning single-path loops with the instruction cache to reduce cache misses during loop execution.

In [79], the authors address two issues with single-path code that we do not address in this paper: (1) it requires special architectural support (like conditional moves) and (2) it increases power consumption since more code is run. They present techniques to address both issues at the cost of a slight reduction in predictability and increased execution time but achieve an increase in power efficiency. In contrast, our work will increase the Patmos processing core's utilization and can, therefore, be expected to increase power consumption.

Trace, superblock, and hyperblock scheduling are all practical approaches to instruction scheduling that are also general enough to work for VLIW architectures [87, 88, 89, 72]. Variations on these techniques can also produce optimal schedules—given enough time and based on some definition of *optimal* [90]. Common to most instruction scheduling approaches is the goal of reducing average-case execution time (ACET). They do this by using profiling (or sometimes other methods [91]) to find the most often-executed paths. They then prioritize reducing the execution time of these paths. This often comes at the cost of the execution time of other, less-frequently executed paths and increased code size. However, they are still effective at reducing the ACET.

For real-time systems, reducing ACET is of little use, so these scheduling techniques have been modified to instead focus on reducing WCET. Similarly to this paper, the authors of [71] investigate how to use very-long instruction word architectures for time-predictability. Even though their work is based on single-path, they limit their practical investigation to basic blocks of innermost loops and do not use loop transformations nor support inter-procedural code. The authors of [73] build on this work, presenting a variation of the hyperblock formation algorithm that takes WCET into account to better select which blocks to merge into hyperblocks. They do this by using a WCET analyzer to identify the WCET path and prioritize reducing its execution time. By doing this iteratively, they refine the schedule to produce code with low WCET. In [74], the authors investigate the same problem except specifically for clustered VLIW processors—a type of processor with many functional units, perfect for code with high instruction-level parallelism. They use a heuristic approach to reducing WCET that uses a precise tail duplication cost model for computing WCET.

Hyperblock scheduling has an inherent conflict with single-path code, as they both use predicated execution. Hyperblock scheduling uses predication to parallelize different paths in the same hyperblock. However, since single-path code is already predicated, the benefit of hyperblock scheduling is removed. Single-path code also has a conflict with the other approaches to WCET reduction. In general, they seek to find the WCET path and reduce its execution time. This will effectively reduce WCET in traditional code at the cost of potentially increasing other paths' execution time because of added compensation code. This is especially notable in tail duplication, which entails duplicating significant portions of a superblock. This is an acceptable trade-off for traditional code as only the worst-case path dictates the overall WCET. However, for single-path code, all paths are executed. Therefore, reducing the worst-case path's execution time by, e.g., 10 cycles, is not acceptable if 100 cycles are cumulatively added to the other paths.[1] The result would increase the single-path code's execution time by 90 cycles, even though the worst-case

[1]Here we refer to other paths through the program graph, before transformation to single-path code.

path is shortened. In contrast to the existing techniques, our approach does not result in any code duplication.

Given the existing techniques' inherent disadvantages when scheduling single-path code, we present an approach that utilizes the characteristics of single-path code for scheduling. Primarily, it relies on the assignment of predicates to instructions and basic blocks to find those that can be scheduled in parallel. Even though our work is only implemented for the Patmos processor, it is general enough to be used for any architecture that supports single-path code.

5.3 Background

This paper presents single-path code optimization for the dual-issue, time-predictable processor Patmos. The following subsection describes Patmos, which has been designed to execute single-path code efficiently. It is followed by a description of single-path code principles and a set of definitions used in the rest of the paper.

5.3.1 The Patmos Processor

Patmos [16] is a processor that is part of the multi-core architecture T-CREST [15]. T-CREST and Patmos aim to build time-predictable computers [80], including time-predictable on-chip communication [92] and memory controllers [93]. Patmos itself is a RISC-style processor optimized for real-time systems. It has two in-order pipelines and therefore avoids any timing anomalies [81]. To simplify the cache analysis, Patmos contains several special caches and a scratchpad memory for data and instructions. Instructions are cached in the so-called method cache [82, 45]. It caches full functions, such that cache misses can only occur at a function call or return. Patmos also contains a stack cache for stack-allocated data [46], which is simple to analyze [94]. Both method cache and stack cache are single-path-friendly cache solutions. Patmos also contains a standard data cache, which is not easy to analyze and not compatible with single-path code's promise of constant execution time. For normal data, we propose using the scratchpad memory or the cache-bypassing load and store instructions.

Patmos supports single-path code with predicated instructions (also called predication). We say an instruction is *enabled* if its predicate is `true` when executed. If the predicate is `false`, we say the instruction is *disabled*. Enabled instructions behave conventionally, going through the pipeline and updating the processor/memory state. However, disabled instructions do not update the processor or memory state. They effectively become `no-ops`. Patmos contains eight predicate bits. Each instruction specifies which predicate bit it will depend on to be enabled or disabled. The instruction also specifies if the bit's value should be inverted before being used as a toggle. Additionally, instructions have the same timing regardless of their predicate's value; a disabled multiplication instruction is not faster than an enabled one. Only updating the processor or memory state is affected by predication, i.e., write-back into the register file or a write into the memory.

Single-path code contains many data-independent instructions. Patmos is a dual-issue architecture, which enables the execution of two such independent instructions in parallel. To implement this parallel execution, instructions need to be scheduled by the compiler and marked as a dual-issue instruction pair, called a *bundle*. The marking of a bundle is a single bit in the first instruction and can be decoded in the fetch stage. This stage uses a split cache (for even and odd addresses) and always fetches two instructions. Both four-byte instructions are used if the first is marked as a bundle, and the program counter is incremented by eight. If not, only the first instruction is used, and the program counter is incremented by four.

5.3.2 Single-Path Code Generation

We call a piece of code *single-path* if its execution enforces the same unique instruction trace, i.e., the same sequence of instructions and accesses to instruction memory for all possible data valuations of the manipulated variables. The point of single-path code is that the enforcement of an invariable sequence of accesses to instruction memory eliminates one of the central sources of timing unpredictability. In particular, when executing single-path code multiple times from the same processor and memory system states (i.e., the same state of the processor pipeline and instruction cache) and when the execution times of instructions are constant, the execution time of the entire code can be expected to be constant.

Constant execution time makes timing repeatable [95], which brings along the following desirable properties:

- It allows for the most precise argumentation about code timing. In the simplest cases—where there is no timing variation from the memory hierarchy—the code will always have the same timing. This makes WCET analysis as simple as running and measuring the code's execution time and produces an exact result.
- When execution times are expected to be invariable, any deviations from the expected timing can be taken as error indicators. Thus, monitoring the execution time of single-path code is a simple but powerful error-detection mechanism.
- An observation of execution times does not provide any clues about the performed computations. This means that single-path code safeguards computer systems against side-channel attacks that use execution-time monitoring to get hints about what is happening in the code. This contributes to computer systems security.

The fact that we use single-path code may seem to limit the applicability of the presented approach to algorithms that do not contain any data-dependent control decisions. Such a limitation is, however, not the case. Single-path code is generated by a compiler that applies special code transformations to eliminate data-dependent control flow from the input source code. Thus, any execution-time-bounded code may be used as a source for

```
if !cond goto Lelse
Lthen:
x = a + 1                ( cond) x = a + 1
goto Lend                (!cond) x = b - 2
Lelse:                                .
x = b - 2                             .
Lend:                                 .
...                                   .
```

Figure 5.1: The difference between branching and predicated execution. On the left, a condition makes the execution skip one of the paths, while on the right, both paths are executed, but only one of them will be enabled at a time.

single-path code generation. Hard real-time code must be execution-time bounded, which means the maximum number of loop iterations and calls to recursive functions must be bounded [83]. Patmos' compiler ensures this by requiring an annotation be added to each loop that specifies the maximum number of iterations the loop can take. If the annotation is not present, a compile error is thrown. An error is also thrown if the code contains any recursive function. The single-path code transformation does not support recursive functions.

Three transformation techniques are needed to create single-path code:

If-conversion

When executing branches, timing variability can be introduced when the two possible alternative paths consume a different amount of time for their execution. To address that, the two alternatives are predicated on the branch condition's value and are both made to execute. Thus, if the condition is true, only the true-path code's predicate is set to true, and vice versa for the other path. The effect is that only the required path is run, but the timing is constant, as both paths' code is executed (with the false path being disabled and therefore not having an effect.)

Figure 5.1 illustrates if-conversion. On the left, we see conventional code that will always branch over one of the execution alternatives. On the right, predicated execution never branches but will instead always disable one of the alternatives.

Nested if-then-else constructs are handled similarly. The various (nested) code blocks are serialized and predicated with different predicates. The predicates for code blocks generated from conditional blocks at deeper nesting levels are computed by combining the originally nested execution conditions of the respective code branches.

Loop-conversion

Loops are another source of timing variability. If the number of iterations taken by the loop changes, the time it takes to execute the loop—and therefore also the program—changes, too. To eliminate this variability, the loop is transformed such that it will always iterate the maximum number of times. However, to maintain the program's semantics, predication is used to disable the loop body as soon as the required number of iterations has been executed. Thus, any superfluous iterations have no effect but are still run to maintain constant timing.

Procedure-conversion

A final source of timing variability comes from the calling of procedures. Even if the execution of a procedure takes constant time, if that procedure is called in some execution scenarios, but not in others, then the program's timing will also be variable, e.g., if the procedure is part of a conditional block whose predicate is `true` for some inputs but otherwise `false`. To maintain constant timing, procedures are called and executed even though the calling code is disabled (e.g., from a disabled path in a branch.) However, all procedures accept an additional predicate-argument used to predicate the entire procedure's execution. If the call stems from disabled code, then the procedure body is also disabled. This conversion ensures that procedures are always called while their functionality follows the predicate of their call context on the execution path. Since each call has constant timing, the whole program will also have a constant execution time.

Transforming code into single-path affects the performance of the final binary. Common among all three of the above techniques is the forced execution of otherwise unused code. The impact of single-path code was studied in [96, 17]. In general, they saw a slowdown below 1.9 times, with some as low as 1.1 compared to the worst case measured. They also saw especially egregious examples around 4 and 5. The "penalty" is heavily dependent on the amount of control-flow in the program. The more control-flow, the higher the penalty. It, of course, also depends on the quality of loop bounds and how single-path-friendly the code is written. The less precise loop bounds are, the more superfluous iterations single-path code is forced to take. Likewise, some control-flow patterns can be optimized to work better for single-path code. One example is code with an if/else statement, where both alternatives call the same function but with different arguments. When converted to single-path, both calls to the function must be executed. However, if the programmer or compiler can instead factor out the function call to after the if/else, they could avoid one of those calls.

5.3.3 Definitions

Basic Block: A sequence of instructions whose execution always starts at the first instruction and may only branch on the last.

Control-Flow Graph (CFG): A directed graph of blocks where the edges model how control flows from one block to the next. A branch is modeled as a block with two out edges in the CFG—one for each path. We do not handle branches with more than two targets. As such, `switch`-like behavior must be converted into a cascade of simple alternatives.

Dominate: A block dominates another block if all paths leading to the latter must first go through the former.

Post-dominate: A block post-dominates another block if all paths going through the latter must eventually also go through the former.

Loop Header: In a loop, the header block is the one that dominates all the other blocks in the loop, i.e., it is the entry to the loop. We associate every block in the CFG with the header of the innermost loop containing it. We also treat the whole procedure as a pseudo loop, where the initial block of the procedure is the header. Therefore, all blocks in the procedure have a header (except the procedure's initial block.)

Back Edge: An edge whose source block is in a loop, and the target block is the header of the same loop.

Exit Edge: Has the source block in the loop but target block outside it. Informally, the edge exits the loop.

Forward CFG (FCFG): An acyclic CFG resulting from removing all back edges from a CFG.

Control Dependence: In a CFG—given the blocks `x`, `y`, and `z`—`x` is control dependent on `y` if `x` post-dominates `z` but not `y`. We also say that `x` is control dependent on the edge `(y,z)`.

Equivalence Class: Two blocks are in the same equivalence class if they are control dependent on the same set of edges.

Guard: A predicate or register guards an instruction if its value determines whether the instruction is enabled or disabled. A block is guarded by a predicate or register if any of its instructions are guarded by the same.

5.4 Single-Path Transformation

The single-path code generation technique takes the CFG of a procedure and rearranges it—with various edits—to produce code with no input-dependent control flow. Our transformation is a variation of the one presented in [17]. In this section, we will describe the transformation informally and highlight the differences from the original work. Section 5.4.2 is wholly our contribution to the single-path transformation, while the descriptions in Section 5.4.1 and Section 5.4.3 are identical to the original work unless where we explicitly state otherwise.

5.4.1 Preparing the CFG

The single-path transformation starts by analyzing the CFG of a procedure. It tracks each loop by its header block and identifies all the remaining blocks that are contained in the loop. This produces a sub-CFG from which an FCFG can be constructed: Two new nodes—`s` and `t`—are added to the graph, with the former connected to the latter and the latter to the loop header. The new nodes model how control flow enters and exits this part of the code. They do not represent any code in the final program and will be removed again in a later step. Then, each block in a loop that has a back or exit edge is connected to `t`.

Take the procedure in Figure 5.2 as an example. For the loop with the header `b`, the transformation will create the FCFG seen in Figure 5.3. Since the whole procedure is also treated as a loop, the FCFG in Figure 5.4 is created for the pseudo loop with header `a`. Here we see how nested loops are only represented by their headers; the blocks `b` and `f` are the headers of the two loops in the procedure.

In traditional code, when an exit edge is taken, the loop no longer executes. Instead, a single-path loop has all its instructions disabled during superfluous iterations. Therefore, non-exit edges are added as control-dependence edges if they are outgoing from an exit edge's source block. This ensures that when exit edges are taken, the header block's predicate is set to false, which disables the whole loop's body for the remaining iterations.

The FCFGs are used to partition the graph into equivalence classes. In our example, we have two equivalence classes: {`a`,`b`,`g`,`h`} and {`f`}. We assign each class a unique predicate that will be used as the guard for its instructions. This is an abstract predicate that is later assigned a physical predicate register when it is used.

So far, the only difference between our transformation and the original is that we track predicates on a per-instruction basis, while the original work tracked it on a per-block basis. Since we will eventually end up bundling blocks, a bundled block would end up with instructions from different equivalent classes. Therefore, we must track each instruction's predicate to ensure the final guard register used is the correct one.

At this point in the original single-path transformation, code generation would continue by reordering the CFG into a straight-line sequence. However, our approach introduces a preceding step for the bundling of blocks. We describe this step in the following subsection, which is wholly part of our contributions in this paper. Section 5.4.3 then continues with the transformation's reordering steps, which are mostly identical to the original work.

5.4.2 Bundling

With the information gathered in the previous step, we can now start bundling blocks together. Conceptually, for any two blocks whose instructions are in different equivalence classes, we can use one issue slot of the Patmos pipeline for the first block's instructions

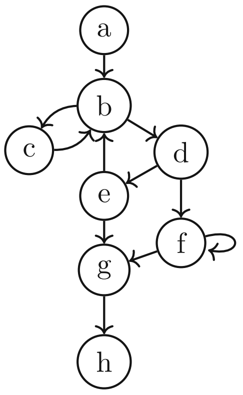


Figure 5.2: A control-flow graph of a procedure.

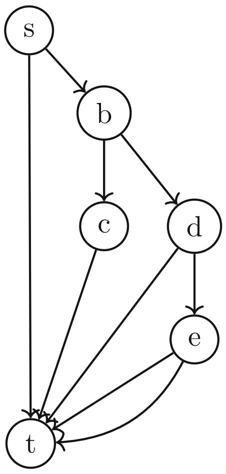


Figure 5.3: FCFG of the loop b.

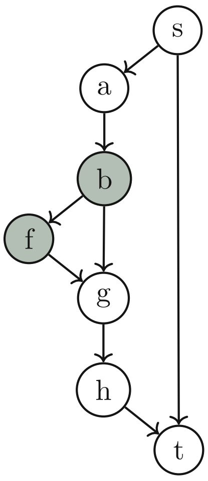


Figure 5.4: FCFG of the pseudo loop a, with the headers of loops b and f in grey.

and the second issue slot for the second block's.[2] In this subsection, we will only describe how to incorporate an algorithm for finding block pairs and bundling them—what we call a bundling algorithm—into the transformation. In Section 5.5, we will then present an exact proof-of-concept bundling algorithm. We have chosen this split to highlight how the transformation technique is independent of the bundling algorithm.

For any two blocks to be bundleable into a combined block, three rules must hold:

- **Non-equivalence:** The pair's instructions must belong to different equivalence classes to ensure that only one block's instructions are enabled at a time. This guarantees that one block's instructions do not interfere with the other's, e.g., when using the same registers.
- **Non-domination:** Neither block can dominate or post-dominate the other. This maintains the semantics of the programs in cases where blocks in different equivalence classes dominate each other. This often occurs, e.g., when one block branches and the other block is part of one of the conditional paths. If they were to be bundled, the code for calculating the branch condition would be executed at the same time as the code for one of the paths.
- **Looping:** They must be in the same loop. This ensures that each block is iterated over the correct number of times.

A bundling algorithm must consists of two parts: (1) `findBlockPair`: Finds a pair of blocks that adhere to the above rules and (2) `bundleBlockPair`: Performs the actual bundling of a pair of blocks, deciding exactly which instructions from each block are paired up, and which one of them goes into which issue slot.

Our single-path transformation continuously calls a bundling algorithm, first getting a pair of blocks from `findBlockPair` and then having `bundleBlockPair` bundle them. This allows the transformation to support a wide range of bundling algorithms. Those that do not store a state between calls are most naturally implemented in this setup, e.g., greedy algorithms. However, algorithms that do need to store a state are free to do so, which means any type of algorithm is supported.

After each call to `findBlockPair` and `bundleBlockPair`, the (F)CFG is automatically reordered such that the semantics of the program are maintained: edges to or from the original blocks are moved to the new bundled block, which maintains the correct control flow. The same is done for predicate definitions. These are used to track how the value of a predicate can be obtained. They are dependent on the condition calculated in a block and are guarded by other predicates. Therefore, we move this information over to the bundled block, such that the next step can emit the correct instructions to evaluate predicates. It is also worth noting that the algorithm is free to pair a block that

[2]Not all Patmos instruction can be issued in the second issue slot, e.g., memory loads and stores can only be issued in the first slot.

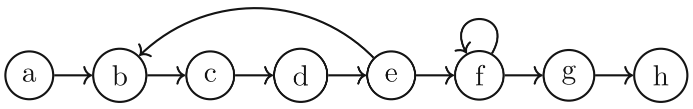


Figure 5.5: The single-path CFG transformed from the one in Figure 5.2 with bundling disabled.

has already been bundled with another block. This means more than two original blocks may end up bundled together into one mega-block, so long as all block pairings adhere to the rules specified above.

The first `findBlockPair` call, which does not return a block pair, signals the bundling algorithms end. The transformation, therefore, moves on to the final code generation.

5.4.3 Code Generation

Each block now has a set of predicate definitions that are depended upon by succeeding blocks' equivalence. Therefore, instructions are inserted that evaluate predicate values and store them in the correct location, be it in predicate registers for immediate use or at spill locations: in general-purpose registers or on the stack.

The next step is to reorder the blocks into a straight-line sequence based on each FCFG. All blocks in the loop are put into topological order—excluding `s` and `t`, which are no longer needed—with each block having an edge to the next block in the sequence. This eliminates all conditional control flow in the loop. Lastly, a branch is inserted from the last block of the loop to the header block (which is always the first in a topological ordering.) This branch is conditioned on the number of times it has already been taken; After N-1 times (where N is the maximum number of iterations the loop should take), the condition becomes false, and execution exits the loop. This ensures the loop iterates exactly N times.

Recall that in an outer loop's FCFG, only the header of a nested loop is represented. To incorporate a nested loop into its parent loop's block sequence, the final block in the inner loop's sequence is connected to the next block in the outer loop' sequence. The result of the single-path transformation on our example procedure—without any bundling—can be seen in Figure 5.5. We can see that the loop with header `b` ends with the block `e`, which is connected to the next block in the outer loop, `f`.

5.5 Implementation

We extend the open-source implementation presented in [17] with support for bundled blocks as described in the previous section. We describe the overall implementation in this section, highlighting what modifications we made to the original implementation. Other than the slight alterations we made to the CFG preparation and code generation, we also added the dedicated bundling pass. This section also describes the implementation of a simple and naive proof-of-concept bundling algorithm incorporated into the bundling pass to be used for our evaluations.

5.5.1 Compiler Overview

The compiler is based on the LLVM compiler framework [49]. LLVM provides a frontend called Clang, which translates C source files into an intermediate representation called Bitcode. This representation is low level, being similar to assembly code but not specific to any hardware architecture. Bitcode is therefore well suited as a target for generic language- and target-independent optimizations, many of which are provided by the framework. The compiler links together the user's code, standard library, and support libraries in Bitcode. This gives the backend implementation a whole-program view for analysis and transformations. This is important for the single-path transformation since it can affect all code—not just the user's code. The backend takes the Bitcode and translates it into machine code for the Patmos architecture.

5.5.2 The Single-Path Passes

The single-path transformation starts with a set of passes that prepare the code for being transformed. The passes ensure certain properties are established before the transformation begins, and in the end, code that needs to be present in both single-path and conventional versions is duplicated. The rest of this section only concerns the single-path versions of the code.

Single-Path Info Pass

This pass analyzes the CFG of each function, finds loops, builds FCFGs, and assigns equivalence classes and predicate definitions to each block. The subsequent passes use this information to bundle and emit the code correctly. The information is updated whenever changes are made to the CFG.

Our work makes two changes to the properties of this pass compared to the original implementation. First, we assign equivalence classes to each instruction instead of each block, as argued for before. We also collect and track predicate definitions beginning in this pass. The original implementation did not track definitions in this pass since it could do it in the Reduce pass described later. However, the information about predicate definitions (gathered from the blocks' instructions) is corrupted whenever blocks get bundled. Therefore, this pass gathers the information. It is maintained throughout block bundling to be used in the Reduce pass.

Bundling Pass

This pass is dedicated to block-bundling, taking an implementation of a bundling algorithm and running it to completion. As a proof-of-concept, we implement a straightforward bundling algorithm:

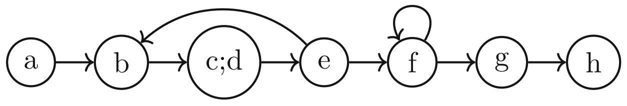


Figure 5.6: The single-path CFG transformed from the one in Figure 5.2 with bundling enabled.

`findBlockPair`: For each loop, we go through all the blocks, looking for any that have two immediate successors or more. When we find one, we look for a successor pair that upholds the following:

- The successor edges are neither back nor exit edges.
- Neither of them is a header of a nested loop.
- Neither of them post-dominates the other.
- Neither of them has already been bundled before.

The first pair of successors to uphold these four requirements is returned by `findBlockPair`. We search through all the loops in the function, starting at each loop header and following a depth-first traversal. If no pair can be found that upholds the requirement, the algorithm is done.

`bundleBlockPair`: We try and bundle the first instruction in the first block with its counterpart in the second block. This is not always possible, as some instructions in the Patmos architecture can only be issued in the first issue slot, e.g., loads and stores from memory have this property. Alternatively, we try to switch them, such that the second block's instruction is in the first issue slot. If this is not possible either, e.g., if both instructions are loads, we simply revert to interleaving them without bundling. We then do the same for the next pair of instructions. If one block has more instructions than the other, we append the additional instructions to the end.

We can show that any pair found by `findBlockPair` is bundleable:

- **Non-equivalence:** If the block whose successors we are bundling—the parent block—is not itself a result of bundling, then it is trivial to see that the successors must be in two different equivalence classes. If the parent is a bundled block, we have two possibilities: (1) Each candidate is the successor of one of the original parent blocks. We know that the two parent blocks are of different equivalence classes, which means their successors cannot be of the same equivalence class. (2) The candidates are the successors of the same parent, which we have already argued for.
- **Non-domination:** We specifically check any candidate pair for whether this rule holds and, if not, disregard them.

- **Looping:** Since we disallow block pairs with exit edges from the parent, we know they cannot be in an outer loop of the parent block's loop. Since we also disallow them being headers of inner loops, the only possibility left is that they are both parts of the parent-block's loop.

Using this bundling algorithm on the CFG shown in Figure 5.2 will result in the bundling of the blocks `c` and `d`, as can be seen in Figure 5.6. Since our algorithm only looks for branching blocks, it considers blocks `b`, `d`, `f`, and `e`. `b`'s successor blocks are the only pair that adhere to our algorithm's rules:

- Neither edge from `b` is a back or exit edge since both `c` and `d` are in the loop.
- Neither successor block is a header of another loop.
- They do not dominate each other.
- They have not been bundled with any other blocks.

For `d`, one edge exits the loop (going to `f`.) For `f`, one edge is a back edge (loop to `f` itself.) For `e`, one edge is a back edge (going to `b`,) and the other is an exit edge (going to `g`.)

At the beginning of the bundling pass, our bundling algorithm's `findBlockPair` is called, which returns the blocks `c` and `d`. Then `bundleBlockPair` is called, bundling the various instructions in those blocks, yielding the bundled block `c;d`. The pass then updates the (F)CFG, such that the new block takes the old blocks' place without changing the semantics. `findBlockPair` is then called again. However, this time no blocks are returned, signaling the bundling algorithm's end and, therefore, also the bundling pass.

Reduce Pass

The reduce pass produces the final single-path code. First, a specialized predicate-register allocation is performed. It assigns predicate registers to each equivalence class. If not enough registers are available, a general-purpose register is used as a spill location, such that the predicate register that is used furthest in the future can be freed for reuse.

This register-allocation algorithm is the same as the original work's algorithm, except modified to handle bundled blocks, which is not much different from handling non-bundled blocks with just one guard predicate.

After predicate-register allocation, all instructions in the function are assigned the physical predicate-register guard that the allocation specified. Function calls are never predicated since the functions need to be run every time. Instructions for spilling or restoring are inserted in newly created blocks and are also not predicated.

Finally, after reordering the blocks into a straight-line sequence, consecutive ones are merged wherever possible, such that the remaining branches are only for facilitating looping.

5.6 Evaluation

We implemented a simple and naive bundling algorithm to prove the single-path transformation's effectiveness in supporting any bundling algorithm. To show that it works and can increase performance, we measure the number of cycles used when executing various benchmark programs with and without bundling (compiled using the `-O2` optimization flag.) The benchmarks are run on a cycle-accurate simulator of the Patmos processor, where all caches are 2-way set associative using a least-recently-used replacement strategy.

5.6.1 Synthetic Benchmarks

Theoretically, a speedup of 2 is the maximum possible when using bundling on the Patmos processor. This occurs when all instructions are paired up in bundles, cutting the number of instructions in half. However, it is practically impossible to reach this ceiling, as there are various reasons inhibiting instructions from being bundled, e.g., arithmetic instructions with immediate values larger than 4095 take up two instruction words on their own, which means no other instructions can be bundled with them. To get a feel for the potential of bundling instructions, we devise three synthetic benchmark programs that are easy for our naive algorithm to bundle.

The first benchmark, `synth_opt`, is designed to be a best-case program when given to our algorithm to bundle. It consists of a loop, whose body is a simple if/else statement with some branchless calculations in each path. This benchmark is simple enough that our algorithm can bundle it optimally. To do so, it merely takes the two blocks that comprise the two branches and bundles them. The branches consist of heavy math calculations to maximize the number of instructions being bundled. We also ensure that the two branches do the same amount of work, such that all their instruction can be paired up, with no spare instructions having to forego being bundled. Lastly, the loop iterates 4095 times. This number of iterations is the maximum possible where the loop counter can still fit into short-immediate arithmetic operations. This is required, as our math uses the loop-counter, which ensures no instructions are optimized away. Any immediate values larger than 4095 are so large the compiler has to put them in a long-immediate arithmetic instruction, which, as stated before, takes up two instruction words and cannot be bundled with other instructions. The result is a program that is primarily a loop with 142 instructions. When bundling is enabled, 132 of the instructions are bundled by our algorithm. The last 10 are the overhead of managing the loop iteration and cannot be bundled by our algorithm. This includes instructions that determine whether to keep looping and calculating the branch condition.

The second benchmark, `synth_if`, is a variation of `synth_opt` introducing nested if/else statements. Each branch of the original if/else statement now contains another if/else statement. Every path in this program is also the same length. This will show us if the algorithm can optimally bundle this slightly more complex example and what performance improvements are attainable with a higher number of execution paths. Similarly to `synth_opt`, the resulting assembly is mainly a loop of 130 instructions.

When bundled, 92 of them are bundled. However, unlike the previous benchmark, it does not get optimally bundled; two blocks that manage the loop increments, among other things, are not bundled, even though they can be. Luckily these two blocks only have two instructions each, so we only missed out on two bundles. The remaining 34 instructions manage looping and predicates. This is a higher amount than in the `synth_opt` benchmark and is caused by the fact that it has additional execution paths. It is also exacerbated by inefficient management of predicate registers, resulting in some predicates being spilled to the stack, which could have been avoided.

The last synthetic benchmark, `synth_asym`, is a variation of `synth_if`. Instead of having inner if/else statements in both branches of the outer if/else statement, the false branch of the outer if/else statement only consists of branchless calculations. This benchmark exercises the algorithm's decision-making, as there are many different ways to bundle the resulting blocks. However, since we have blocks of different sizes—the two branches in the inner if/else statement individually contain fewer calculations than the false-branch of the outer if/else statement—some bundlings are better than others. Looking at the assembly, it comprises a loop of 114 instruction, where only 48 get bundled. To understand this low amount of bundling, we will have to take a look at some of the blocks in this programs:

1. Calculates the condition of the outer if/else statement.
2. The body of the false-branch of the outer if/else statement. This is the largest block by far.
3. Calculates the condition of the inner if/else statement in the true-branch of the outer if/else statement.
4. The true-branch of the inner if/else statement.
5. The false-branch of the inner if/else statement.

Block 1 cannot be bundled with any other block, as they are all control dependent on it. Blocks 4 and 5 cannot be bundled with block 3 for the same reason. Any of blocks 3-5 can be bundled with block 2, as they are in different branches of the outer if/else statement. Our algorithm chooses to bundle block 2 with block 3. This is a natural choice, as those two blocks are both direct successors of block 1. However, this is the worst choice, as block 3 is significantly smaller than 4 or 5. The optimal choice, which our algorithm cannot perform, is to bundle block 2 with both blocks 4 and 5. This is possible because block 2 is slightly larger (by design) than those blocks combined. They could have been scheduled in the second issue slot directly after each other, while block 2 would take up the first issue slot.

Table 5.1 shows the number of clock cycles used to execute each benchmark program without bundling (*Original*) and with bundling (*Bundled*). It also shows the difference between the two counts and the resulting speedup.

Table 5.1: Execution time (in cycles) for the various benchmark programs with or without bundling enabled. The first group of programs is the synthetics, then the TACLe benchmark suite, and lastly, the drone control/estimation functions.

	Original	Bundled	Difference	Speedup
synth_opt	586 735	316 399	270 336	1.854
synth_if	521 160	349 149	172 011	1.493
synth_asym	471 993	373 689	98 304	1.263
binarysearch	2 975	2 965	10	1.003
bsort	428 418	428 218	200	1.000
countnegative	43 321	42 439	882	1.021
adpcm_dec	7 173 323	7 173 317	6	1.000
adpcm_enc	7 191 612	7 191 600	12	1.000
control	15 651 568	15 688 468	-36 900	0.998
estimation	14 865 814	14 851 716	14 098	1.001

For our `synth_opt` benchmark, we can see that without bundling, the program requires 586 735 cycles to execute, while it only requires 316 399 cycles with bundling. This is a speedup of 1.85.[3] The relative reduction in clocks used almost exactly matches the reduction in the number of bundles in the loop. Since 132 instructions were bundled together, 66 cycles were removed from the original 142 cycles required to run the loop. This is a reduction of roughly 46,5%. The last half percentage point not reflected in our results can be attributed to the instructions run before the loop even starts.

For the `synth_if` benchmark, we can see that the improvement only amounts to a speedup of 1.49. This lesser improvement can be attributed to the higher amount of predicate management instructions that are not being bundled. For the `synth_asym` benchmark, we see a speedup of 1.26, which cements the importance of bundling blocks intelligently, unlike what our algorithm does here.

5.6.2 TACLe Benchmarks

We also evaluate our algorithm's performance on 13 programs from the TACLe benchmark suite [50]. We cannot use all the programs in the suite. First, some use recursion, which is not supported by the single-path code transformation, while others have loops without valid loop bounds, which is also a necessity for single-path code. Additionally, our compiler currently fails to generate single-path code for many of the rest of the benchmark's programs. Therefore, we only evaluate the execution times of 13 of the programs in the benchmark that were successfully compiled using single-path code.

Only 5 of the 13 programs measured resulted in bundles being created when bundling was enabled. These are, therefore, the only programs shown in Table 5.1. This is caused

[3]Notice that a speedup of 1 means no change to the execution time. A value below 1 is an increase in execution time and therefore a slowdown. A speedup of 2 is the theoretical maximum.

by the bundling algorithm's simplicity, which is not sophisticated enough to see bundling opportunities in the remaining programs. In Table 5.2, we can get a more detailed look at what the algorithm is doing. It shows compiler statistics regarding the bundling of blocks for each of the benchmark programs we have run. First are the number of basic blocks in the program (*Before*), then the number of pairs of blocks the algorithm found suitable for bundling (*Pairs Bundled*), and lastly, the pairs it did not find suitable (*Pairs Rejected*). By "pairs," we mean two blocks with the same parent and are then checked for whether they can be bundled. As an example, we can look at the `synth_asym` benchmark. It originally had nine blocks in its code. Two pairs of blocks were bundled, which corresponds to blocks 2-3 and 4-5, while one other pair was rejected. Looking at the five TACLe programs with blocks bundled, we can see that not many block pairs were bundled. `countnegative` was the benchmark that resulted in the highest relative performance increase, even though this is based on only one block pair being bundled. The `adpcm_dec` and `adpcm_enc` did slightly better by bundling two pairs each. However, since they are much larger programs, it resulted in a negligible performance increase.

We also take a look at whether `bundleBlockPair` found the optimal instruction bundling. A sub-optimal bundling would revert to interleaving instructions instead of bundling them. Table 5.3 shows the total number of instructions in the programs (*Before*), then the number of instruction pairs that were bundled (*Pairs Bundled*), and lastly, the number of instruction pairs that should have been bundled but could not because they both needed to be in the first issue slot (*Pairs Interleaved*). We can see that all instructions that could have been bundled were bundled. Therefore, a better algorithm for `bundleBlockPair` would have made no difference.

In general, we can see that bundling does increase performance, though, as we expected, not by much. For most of the programs with bundled blocks, the increase is negligible. The exception is *countnegative*'s 1.02 speedup which is similar to the proportion of bundles created, 2, from the 79 theoretical maximum. This is, therefore, in line with expectations.

5.6.3 Real-World Use Case

To close out the evaluation of our bundling algorithm, we use a real-world program that would be a suitable use case of bundled single-path code. We measure the execution time of the state estimation and control functions presented in [18]. These are used for the automatic control and stabilization of a drone. This drone platform is specifically engineered to use real-time hardware and software components to ensure the drone's correct flight. We measure the functions on a real-world data set, acquired from a drone's test flight, and later inserted into the program to be provided to the functions during the benchmark. The benchmark program consists of a loop that first loads the drone's sensor data, passes it to the state estimation function, and lastly calls the control function to perform the correct flight adjustments. This loop is executed 1024 times (with the same number of data points), and we measure only the execution times of the two relevant functions.

Table 5.2: Statistics on the number of blocks in each compiled program.

# Blocks	Before	Pairs Bundled	Pairs Rejected
synth_opt	6	1	0
synth_if	12	3	0
synth_asym	9	2	1
binarysearch	9	1	0
bsort	17	1	1
countnegative	15	1	0
insertsort	20	0	4
jfdctint	14	0	0
matrix1	22	0	0
md5	55	0	17
adpcm_dec	60	2	6
adpcm_enc	64	2	8
h264_dec	43	0	4
petrinet	163	0	124
duff	8	0	0
test3	526	0	0
control	39	12	21
estimation	13	2	4

Table 5.3: Statistics on the number of instructions in each compiled program. Omitted the 5 TACLe programs that produced no bundles.

# Instructions	Before	Pairs Bundled	Pairs Interleaved
synth_opt	187	33	0
synth_if	174	23	0
synth_asym	168	12	0
binarysearch	117	2	0
bsort	147	3	0
countnegative	159	2	0
adpcm_dec	1 261	2	0
adpcm_enc	1 471	2	0
control	2 179	33	6
estimation	1 737	2	0

Table 5.1 shows the control function (`control`) and the state estimation function (`estimation`). We see a slight slowdown for `control` and a slight speedup for `estimate`. However, as seen in Table 5.2 and Table 5.3, the algorithm did find small bundling opportunities for both functions. The reduction in `control`'s performance is caused by a slightly worse cache performance when bundling is enabled. The method cache consistently missed one more time in bundled code than in non-bundled code (which had 12 misses.) This extra cache-miss resulted in about 0,7 % more cycles being spent waiting on the method cache in bundled code. This is probably caused by the bundled code being slightly larger than the non-bundled. The data-cache also fared worse, where the miss rate was intermittently higher for the bundled code. However, the data cache is used much less than the instruction cache, with the bundled code only missing once or twice more per call. In general, the slight increase in performance gained from the 33 bundled instruction pairs was heavily offset by the worse cache performance. We also see that the algorithm performed sub-optimal bundling in `control`, as six instruction pairs were forced to be interleaved. Though, even if this had been done better, the effect would be negligible.

To verify the cache misses causing the reduction in performance, we rerun the non-synthetic programs with the memory system burst size increased to 32 bytes (the default is 16 bytes, which was used for Table 5.1) using `pasim`'s `-bsize` flag. This causes cache misses to have a lower penalty, as data is loaded faster. As seen in Table 5.4, this change reduces `control`'s execution time to 9 409 759 cycles without bundling and 9 360 559 with bundling. This means bundling now produces a speedup of 1.005. Note that the number of method cache misses does not change with an increase in the burst size. We can decrease the number of misses by enlarging the method cache, but that does not significantly change the speedup of `control`, as the bundled code always misses more than without bundling. Changing cache sizes did not have a significant effect on speedups. We also see a slowdown of `estimation` when using the larger burst size. This is caused by a slightly worse utilization of the method cache, where more bytes are loaded from main memory that need to be discarded. This causes more stalls in the method cache, even though no extra cache misses occur. For example, when not bundled, the first run of the function required the method cache to free 1 920 bytes, while the bundled version needed to free 1 936 bytes. This results in the stall count increasing from 5 880 to 5 943.

Source Access

Patmos and the T-CREST platform are available as open-source and include the contributions of this paper. The Patmos homepage can be found at http://patmos.compute.dtu.dk/ and provides a link to the Patmos Reference Handbook [51], which includes build-instructions in Section 5.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-

Table 5.4: Execution time (in cycles) for the various benchmark programs with or without bundling enabled using a memory system burst size of 32.

	Original	Bundled	Difference	Speedup
binarysearch	2 660	2 650	10	1.004
bsort	427 788	427 609	179	1.000
countnegative	41 830	40 990	840	1.020
adpcm_dec	7 169 543	7 169 558	-15	1.000
adpcm_enc	7 181 490	7 181 394	96	1.000
control	9 409 759	9 360 559	49 200	1.005
estimation	9 054 295	9 104 772	-50 477	0.994

llvm.[4] However, we advise following the handbook instructions to correctly set up a machine to build and run Patmos programs. The synthetic benchmarks can be found at https://github.com/t-crest/patmos-misc.[5]

5.7 Conclusion

In this paper, we have presented a single-path code generator that leverages the Patmos architecture's dual-issue pipeline by using its second issue slot. We implemented the generator in the Patmos processor's compiler and showed that it could incorporate a wide range of bundling algorithms.

We have implemented a simple bundling algorithm that takes single-issue single-path code and produces dual-issue single-path code. We evaluated the work by first devising a set of synthetic benchmarks to investigate the potential speedup of using bundled code. This showed that doing bundling well can give good results, but doing it sub-optimally can quickly reduce the benefits. We also used a subset of the TACLe benchmark suite and a real-world drone control program to see how our algorithm performs in representative scenarios. In general, the algorithm provided little to no real-world benefit. This was expected as the algorithm is meant as a simple proof-of-concept to show the potential impact of using the second issue slot of the Patmos architecture.

More work is needed to bundle instructions more effectively. A more sophisticated bundling algorithm is needed to effectively find blocks to bundle for maximizing the use of the second issue slot. To do this, the algorithm should analyze more than just control-flow. For example, it could look at the resource utilization of blocks and pair those blocks that don't contend with each other. Another example could be finding very similar blocks, which could allow it to merge identical instructions into one instruction used for both

[4]The code implementing the bundling can be found in the file: `lib/Target/Patmos/SinglePath/PatmosSPBundling.cpp`. Commit Hash: 55c7a000393dded7a5886c5fcb6c665d95b5fa7f

[5]Subfolder: `experiments/singlepath_vliw/synthetic_benchmarks`. Commit hash: da0ff6bcebf0e5acc15c1ee1ee4e5889720aefe2.

blocks. Further work is also needed to make single-path code more predictable. Using the data cache introduces variance into single-path code's execution that requires WCET analysis. We plan on tackling this issue such that single-path code executes in constant time, even when accessing memory through the data cache. This will need a specially designed cache that allows more fine-grained control of what data is stored where and for how long.

CHAPTER 6

Two-Step Register Allocation for Implementing Single-Path Code

Abstract—*Register allocation is a crucial step in the compilation pipeline that decides what program values occupy which physical registers. Single-path code's use of predicated instructions instead of branching control-flow means register allocation must also allocate predicate registers. In this paper, we improve the original single-path transformation to allow generic register allocators to allocate predicate registers. Our improved transformation splits register allocation into two. First, the general-purpose registers are allocated as usual using a generic register allocator. Then, the main steps of the single-path transformation are performed while still using virtual predicate registers. Lastly, register allocation is rerun using the generic allocator to allocate the predicate registers. Our results show the improved single-path transformation increasing performance by up to 80 % and reducing code size by up to 43 % compared to the original transformation that uses a custom predicate allocator.*

6.1 Introduction

Real-time systems must have static guarantees on the worst-case execution time (WCET) so that they can react to their environment in time to avoid catastrophic failures. Traditionally, a WCET analysis tool is used to get a conservative estimate on the WCET, a bound. Single-path code is an alternative to the use of such tools. It transforms traditional code, which uses branching control flow, into a single sequence of instructions. This removes any execution variability from control flow, which, coupled with the proper hardware, results in programs with constant execution time. Thus, using single-path code negates the need for a WCET analyzer. However, current implementations of single-path code suffer from sub-optimal performance, which allows analyzers to find lower WCET bounds than the execution time of single-path code [69].

To avoid using branching control flow, single-path code uses predicated instructions. Such instructions are *enabled* and execute normally when their additional predicate operand evaluates to true. When the predicate is false, the instruction is *disabled* and has no effect. It does not update any registers or alter any memory. Predicated instructions can be substituted for branching control flow by using the branch condition as the predicate for one branch path and the negated condition for the other. Instructions in all paths are executed, with only the relevant one having any effect.

Predicates add a register class that must be managed by the compiler. Register allocation is a crucial step in the compilation pipeline that decides which program values occupy which of the limited number of physical registers at a given point in the program. Predicate registers (PRs) significantly affect register allocation because they decide whether instructions are effectual or not. Disabled instructions do not read or write their non-predicate operands, meaning a register allocator needs to use that knowledge in its allocation for maximal performance. Traditional allocators cannot do so, meaning a different approach must be taken to ensure all register classes are allocated efficiently. One such approach is to build dedicated allocators with first-class support for predicates [97]. However, that is a big task best not undertaken lightly or halfheartedly.

In this paper, we present a new single-path transformation that improves register allocation performance while reusing generic allocators otherwise incapable of efficiently allocating predicate registers. Our allocation technique first allocates the general-purpose registers (GPRs), after which the single-path transformation runs using virtual predicate registers. The generic allocator is then rerun, this time only allocating predicates with the code in single-path form. This two-step allocation allows the GPRs to be efficiently allocated in the presence of branching control flow, while the PRs are allocated after branches are removed.

Our contributions are: 1) a two-step register allocation technique that uses a generic register allocator to allocate predicates efficiently, 2) an in-depth description of the single-path transformation using two-step allocation and 3) an open-source implementation of the improved single-path transformation.

This paper is organized into eight sections: The following section presents related work. Section 6.3 provides information on the Patmos architecture, register allocation, and single-path code. Section 6.4 outlines why a generic allocator must use two-step allocation for efficient GPR and PR allocation. Section 6.5 describes the original transformation and its challenges. Section 6.6 describes how the improved transformation augments the original to use the generic allocator for predicates. Section 6.7 evaluates the impact of using the improved transformation compared to the original. Section 6.8 concludes.

6.2 Related Work

Predicated instructions add considerable complexity to the compilation process. Algorithms, analyses, and optimizations must be updated for disabled instructions. Examples include constant propagation [98], data-flow analysis [99], and partial redundancy elimination [100]. Traditionally, intermediate representations like LLVM do not model predicated instructions directly [49]. However, some work has explored modifying static single assignment [101]—often the basis for intermediate representations—to also model predication [102, 103].

Tracking the relationship between predicates is essential to ensure algorithms and optimizations can account for when instructions can and cannot affect each other [104]. Therefore, work has been put into managing and using such information. For example, boolean expressions can be maintained for each predicate specifying which paths would result in it being true [99]. The expressions can then, e.g., be used to determine if two predicates can be true simultaneously. Predicate-aware scheduling leverages this information to allow multiple instructions to be simultaneously scheduled for the same functional hardware units as long as only one is enabled at run-time [105]. Single-path code tracks relationships between predicates using equivalence classes and leverages it for more efficient scheduling.

Predication is traditionally leveraged by doing if-conversion, where multiple paths with branches are merged into one using predication instead of branching [106]. The result is a basic block with no branching. A relevant question is when in the compilation pipeline if-conversion should be performed. The authors of [107] chose a two-stage approach, where if-conversion is performed early in the compilation process while potentially reversing some of it during instruction scheduling. In this work, if-conversion is part of the single-path transformation performed between our two register allocation steps.

A variant of if-conversion makes a *hyperblock*, a set of predicated basic blocks where control always enters at the top but may have multiple exit points [108]. This allows better optimization by not forcing all predicated paths to be executed every time. Hyperblocks and if-conversion have been found to increase power significantly because more code is executed; however, it does not always result in higher energy use overall [109]. Other work has leveraged hyperblocks to reduce WCETs by optimizing the worst-case paths first, sometimes to the detriment of other paths [74]. Such work does not translate to single-path code, as the potential increase in execution time of shorter paths might overshadow the reduction of longer paths. Since single-path code executes all paths, the cumulative execution time might increase instead of decrease.

Register allocation is one of the most critical steps in the compilation process and has received significant attention. It is a challenging problem to solve, as it has been shown to be NP-complete [110]. However, the sub-problem of identifying whether any register spilling is needed can be solved in linear time [111]. Work has also focused on dedicated predicate-aware register allocators [112, 113]. Such allocators could be used as an alternative to our two-step approach using generic allocators. However, it would require significant effort to implement such an allocator from scratch, and it would probably need to be updated to support single-path code, as previous approaches do not consider cyclic predicated code [97].

Many different allocation approaches have been used to balance speed and performance. The most well-known algorithm is graph coloring [114]. A graph is constructed where nodes represent values and edges connect values that are live simultaneously, meaning they cannot occupy the same register. The graph is then colored such that no edge connects same-colored nodes. Because the colors represent the available physical registers, successful colorings map values and the physical registers they should occupy. Since graph coloring is NP-complete, register allocation must be as well [115]. Another approach is linear scan, where variables are allocated to registers greedily from the start to the end [116]. It is much faster than graph coloring but has slightly poorer performance. The allocation problem can also be modeled as a puzzle where all the pieces (values) need to fit on a checkered board (registers) [117].

LLVM is a compiler framework that makes it easy to write new compilers and target new architectures [49]. LLVM's register allocator initially used linear scan [118]. However, the practicalities of making a generic register allocator that multiple target architectures can use forced the project to switch to a pair of allocators called *Basic* and *Greedy*. Their new approach mimics linear scan but allocates the values based on a priority queue of spill weights (an estimate of the performance cost of moving a value to the stack) [119]. Further work has improved the allocators to increase the use of compressed instructions like those of the RISC-V architecture [120]. The authors of [121] presented a register allocator based on machine learning. While they outperformed LLVM's Basic allocator, they were not as successful when compared with the Greedy allocator. LLVM's allocators can allocate predicate values to PRs but do not account for disabled instructions. Our work uses LLVM's Greedy allocator and compensates for its lack of predicate support by running it twice.

Single-path code can be generated from any WCET analyzable source code [13]. The authors of [17] implemented the first compiler to generate single-path code. It used two register allocators: LLVM's generic allocator and a custom predicate allocator. We alter this initial implementation to use the generic allocator exclusively. Work has also looked at ensuring constant execution times on systems with multi-cycle memory access latency [69]. Most modern systems likely fall into this category, making the described compensations a must for avoiding the need for a WCET analyzer. Previous work has looked into leveraging single-path code's ILP for increased performance [19]. In the middle of the single-path transformation, knowledge about control flow was used to easily find basic blocks that can be merged, such that each leverages one issue slot of a dual-issue pipeline. Single-path code requires specific hardware support, which can be hard to come by. Therefore, work has been presented that can allow single-path code to run on otherwise unsupported processors through an instruction filter [20].

6.3 Background

6.3.1 Patmos

The Patmos instruction-set architecture (ISA) was intentionally crafted for real-time systems, ensuring time predictability and optimizing for a low WCET [16]. Every Patmos instruction is associated with one of eight PRs for enabling or disabling the instruction. All instructions in the Patmos ISA are predicated. Each instruction also has a predicate negation flag. When the flag is set, the instruction is enabled when its predicate register is evaluated as false and vice versa. For example, take the instruction `add (p1) r1 = r1, r2`. It is an arithmetic addition instruction predicated on the `p1` predicate register. To negate the predicate, an exclamation mark would be added (i.e., `!p1`).

Patmos also features dedicated instructions for manipulating predicate registers [51]. This includes arithmetic operations like `pand` and `por` that implement the logical AND and OR operations. There are also instructions for producing predicates. The most obvious are the comparison instructions, which put their boolean results in a PR. Other instructions can also move predicates to and from the GPRs. The `btest` instruction moves a specific bit from a GPR to a PR, while the `bcopy` moves a predicate to a specific bit in a GPR.

A unique characteristic of Patmos is its split caches, where the traditional data cache is split into a data and a stack cache [46]. The latter can be directly manipulated to always contain the relevant function-local data, ensuring accesses to it never miss. Dedicated instructions can perform accesses that always hit the stack cache. For example, `lws` loads and `sws` stores one 4-byte word.

6.3.2 Register Allocation

ISAs have a limited number of registers available for use by a program. For example, Patmos has eight PRs (`p0-p7`) and 32 GPRs (`r0-r31`) [51].[1] We call these the *physical registers*. In contrast, programs produce many more values than available registers and often have many more values active simultaneously than an ISA will have physical registers. Early in the compilation pipeline, compilers often ignore register limitations and assign values to *virtual registers*, of which there is no limit. At some point, virtual

[1]Patmos also has special-purpose registers (`s0-s15`), but they do not need allocation.

registers must be substituted with physical registers. Register allocation decides at each point in the program which virtual register maps to which physical register [122].

When more virtual registers are in use than the number of available physical registers of the same type, we must offload some values to memory. One of the primary responsibilities of a function's stack frame is to help alleviate the need for physical registers. When the allocator determines that the registers are better spent holding other values, a value can be *spilled* to the stack from a register. The position in the stack used for storing a value is called a *spill slot*. Analogous to registers, different values may occupy the same spill slot at different program points as long as they do not do so simultaneously. When a value residing in a spill slot is needed again as an instruction operand, it is *restored* by moving it back to a physical register.

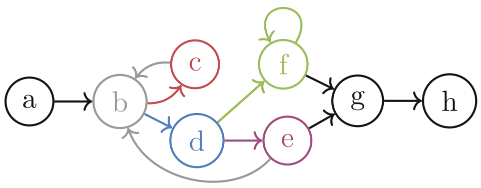
The register allocator is tasked with determining which virtual registers should be spilled and restored at specific points in the program [123]. It aims to minimize the cost incurred from spilling and restoring. Take graph coloring as an example. The allocator might determine that it is impossible to color the graph without having an edge between same-colored nodes. This means assigning the physical registers is impossible without having at least two virtual registers assigned to the same physical register. The allocator looks for a suitable virtual register that is unneeded and spills it, updating the graph to reflect the reduced number of live virtual registers. It then tries to color the graph again, repeating until a coloring is found, which is converted into a mapping between virtual and physical registers.

6.3.3 Single-Path Code

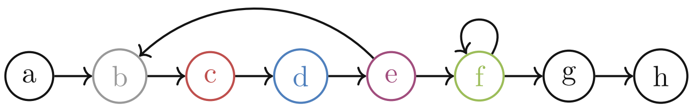
Single-path code uses predication to convert the branching control flow of a function into an instruction stream with only one path [13]. That removes all data-dependent execution-time variability from the instruction stream. Several techniques convert a program into single-path code and ensure constant execution time [17]. Here, we will only cover if-conversion and loop-conversion:

If-Conversion

We need to convert any conditional branching into a sequence of predicated instructions such that only the needed path's instructions are enabled at run-time. The resulting code always executes all instructions in both paths. However, only one path's instructions are enabled at any given time. Looking at Figure 6.1, we can see the result of transforming a program into single-path code. The basic block b conditionally branches to c or d in Figure 6.1a. The color coding of Figure 6.1a's blocks matches the conditions that led to that path being taken. In Figure 6.1b, the colors indicate that only if the corresponding condition is true at runtime will the block's instructions be enabled. As such, we can see how if-conversion results in b always leading to first c and then d. However, only if the red condition holds at run-time will c's instructions be enabled. The same holds for d, leading to either e or f in the traditional code but eventually leading to both in the



(a) Traditional



(b) Single-Path

Figure 6.1: Conversion of a program with branching control flow (top) to single-path (bottom).

single-path version. Notice how we have not colored the edges in the single-path version, as they are always taken.

Loop Conversion

Loops may iterate a variable number of times depending on run-time conditions. Single-path code converts loops to always iterate the maximum possible number of times. Any superfluous iterations are disabled using predication. The result is invariable iterations while maintaining program semantics. The minimum and maximum iteration counts are usually given in the source code as an annotation on the loop. Our example shows that the program has two loops, one containing blocks b, c, d, and e, and the second containing only f. The single-path loop counts how many iterations are executed and keeps looping until the maximum is reached. Inside the loop, the condition that traditionally breaks out of the loop is instead used as the predicate to all the instructions. This condition will become false at some point, meaning any further iterations will have disabled instructions.

6.4 Incompatible Allocation Needs

Single-path code has two types of registers needing allocation: the GPRs and the PRs. During register allocation, the control-flow graph (CFG) is analyzed to see how many values are in use at each point in the execution. An essential part of traditional allocation is noticing that if we have two mutually exclusive paths, we can safely use the same physical registers in both paths, as only one path is executed at run-time. For example, the blocks c and d in Figure 6.1a are on different paths and will never interfere with each other, meaning they can use the same physical registers without issue. If we wait with register allocation until after the single-path transformation, we would end up with the linearized CFG seen in Figure 6.1b. The generic allocator would, therefore, lose information about mutually exclusive code paths—because they are now controlled by predication instead of control flow—and would not be able to reuse physical registers between mutually exclusive paths (because it only sees one path.) In Figure 6.1b, it would see that c precedes d and so would think it cannot touch any registers in c that d may depend on from b. This severely and needlessly increases the amount of spilling and restoring of registers, as the GPR pressure (the number of physical registers needed at a specific point in the execution [124]) becomes affected by all paths simultaneously instead of each path individually. Therefore, a generic register allocator can only produce acceptable performance if the GPRs are allocated before the single-path linearization.

The allocation of the PRs has the opposite problem to the GPRs. Whether a given execution path is enabled at run-time depends on the value of its assigned predicate. For two exclusive paths, two predicates are used, where at most one is true at a time. However, crucially, both predicates may be in use simultaneously. For example, in the case of the paths $b \to c$ and $b \to d$ in Figure 6.1b, their predicates are initialized in b, meaning they are both live in c. This means the two predicates must use different physical registers. However, if we allocate the PRs before linearization, the generic allocator will use the control flow to decide when a physical PR can be reused between paths. It might assign the same physical register to two paths, meaning one would lose its correct predicate value. This does not happen after linearization, as the generic allocator will only see one path and know not to reuse physical PRs. As such, using a generic register allocator can only result in correct allocations if the PRs are allocated after the single-path linearization.

The competing needs of the GPRs and PRs and our aim to reuse a generic register allocator require us to do a two-step register allocation. The idea is to have the generic allocator first allocate the GPRs, perform the single-path transformation, and then force the generic allocator to run again, now with only the PRs being affected. LLVM and its generic allocator are not built to run twice, which is reflected in our implementation steps, but they can be made to do so with the proper coaxing.

6.5 The Original Transformation

The original single-path transformation structure was presented in [17] and extended in [69]. This section will present it again in greater detail, focusing on enabling us to showcase the differences in the improved transformation.

6.5.1 LLVM-IR Preparation

Before converting a CFG into single-path code, it is prepared to meet the requirements of subsequent passes. The Clang front-end produces LLVM-IR format that the LLC backend ingests for conversion to machine code. While still in the LLVM-IR format, three preparatory steps are performed:

S.1 Unifying Function Exits: The CFG is transformed such that there is only one block from which the function returns. We call this (now unique) block the *end* *block*. This is done to ensure there is no need for predicate manipulation in the end block, as the block is guaranteed to be executed. This is the only way to ensure that we can reset the Patmos special-purpose register `s0` without worrying about the correctness of the end block. The `s0` register contains all the PRs and is callee-saved, so must be returned to its initial state before returning from the current function [51].

S.2 **Switch Lowering**: LLVM-IR has first-class support for switch statements. E.g., the instruction:

```
switch i32 %x, label %b1, i32 1, label %b2, i32 2, label %b3
```

will branch to the block `%b2` if `%x` evaluates to 1, it will branch to `%b3` if it evaluates to 2, and otherwise branch to the default target `%b1`. There may be any number of non-default targets. To simplify if-conversion, switch-instructions are substituted by a chain of simple, two-way branches.

S.3 **Function Cloning**: LLVM requires all machine code functions to have a parent LLVM-IR function. Because we do not yet know which functions need to be in traditional form, single-path form, or pseudo-root single-path form, we clone each function twice, once for single-path and once for pseudo-root single-path. Pseudo-root functions are specially optimized because they are guaranteed to be called a fixed number of times [125].

After these three steps, the default LLVM instruction selection passes take over to convert the LLVM-IR function into machine code form. This form is not exactly machine code but is close to it, with only a few pseudo-instructions present. It will be refined to match machine instructions exactly and is what the rest of the single-path transformation works on.

6.5.2 Machine Code Preparation

In the initial machine code form, functions use virtual registers and static single assignment form. This form ensures all registers are only assigned an initial value and never overwritten. Up to four steps are taken to prepare the code for single-path transformation:

S.4 **Data Cache Elimination**: When constant execution time code is requested, we cannot use the data cache to access memory [69]. The conversion from LLVM-IR to machine code produces accesses to data through the data cache by default; e.g., `lwc r1 = [r2]`. Every such instruction is converted to an equivalent one, which directly performs the same access directly to main memory; e.g., `lwm r1 = [r2]`. This step is skipped if constant execution time code is not requested through the flag `--mpatmos-enable-cet`.

S.5 **Function Marking**: Step **S.3** clones all functions in the program without regard for how they are used. This step traverses the call tree from all single-path roots to remove unneeded function clones. While traversing, all reached functions are marked for whether they are needed in single-path code as pseudo-roots, non-pseudo-roots, or both. The clones of each function that are not needed are subsequently deleted. For example, a function only needed as a pseudo-root has its non-pseudo-root clone deleted.

S.6 **Predicate Preallocation**: Managing the PRs in the original single-path transformation is done by the transformation itself. This contrasts with the GPRs, managed mainly by the generic LLVM register allocation passes. The generic allocator also manages the register spill slots on the stack. Therefore, we must determine how many predicates we will be spilling and restoring in the eventual single-path code. We then encode how many spill slots the predicates need, such that the generic allocator makes room for them on the stack so predicate slots do not overlap with general-purpose slots.

S.7 **Memory-Access Compensation**: When constant execution time code is requested, we must ensure all functions access memory the same number of times at run-time. This step first analyses how many memory accesses are executed at minimum and maximum and chooses a compensation technique that most efficiently compensates for any variation [69]. If the decrementing-counter compensation (DCC) technique is chosen, it is immediately performed. If the opposite-predicate compensation (OPC) is chosen, the function is labeled as such so that Step **S.9** can later perform the compensation. OPC must know which predicates are assigned to each instruction to use the same predicate in the compensation [69]. At this point, no assignment has been made, so OPC must be performed later.

The code is now ready for LLVM to run its generic register allocator. The code at this point contains mostly uses of the GPRs, with PRs only used by comparison instructions followed by conditional branch instructions. After register allocation, LLVM inserts the function prologue and epilogue. The former is a code sequence at the start of a function that sets up the stack frame and saves any required registers before the function runs, as per the Patmos application binary interface (ABI) [51]. The latter is a code sequence that restores the stack frame of the caller and any saved registers to their initial value. As mentioned earlier, this is where `s0` is saved and restored. The prologue and epilogue code is not materially affected by the single-path transformation, except where the single-path code affects the ABI. E.g., the original transformation reserves the `r26` registers for its use, meaning the prologue and epilogue must spill and restore it. `r26` is reserved for two uses: First, as a loop counter, ensuring that a loop iterates the same number of times. The register must be reserved because register allocation happens before the main single-path transformation. Without a reserved register, the transformation might be unable to find a free register and would have to spill and restore registers. The second use of `r26` facilitates PR spilling. Since predicates only take up one bit, a general-purpose stack spill slot can store 32 predicates. Therefore, when a predicate needs to be spilled from the PRs, the relevant 32-bit spill slot is loaded into `r26`, updated with the predicate value, and spilled back to the stack.

6.5.3 Main Transformation

The bulk of the single-path transformation is now ready to be performed using six steps:

S.8 **Equivalence Classification**: The control flow is analyzed to find the critical edges resulting in each basic block's execution [17]. The blocks are grouped into equivalence classes, representing that they will be executed under the same conditions. Classes are assigned unique virtual predicates that will be mapped to physical PRs later. Predicate definition points are also identified in this step to guide how predicates are managed in Step **S.11**.

S.9 **OPC Compensation**: If Step **S.7** decides that a function should use the OPC compensation technique, this step performs it. A compensatory load instruction is added using the negated predicate for every memory-accessing instruction. This guarantees that precisely one of the original or compensatory instructions is enabled at run-time, removing any execution time variation from accessing memory.

S.10 **PR Allocation**: A custom register allocator takes the equivalence class information and performs register allocation for the PRs. Each virtual register from Step **S.8** is assigned a physical register to use at a specific point in the CFG. Because loops are almost self-contained, we know any predicates from equivalence classes outside the loop will not be used again until the loop terminates. Therefore, the custom allocator recognizes when a loop needs all physical predicates and spills the `s0` register before the loop starts, avoiding having to spill each predicate individually.

S.11 **Predicate Application & Definition**: The register allocation updates every instruction with the physical register assigned to its equivalence class. Then, the predicate definition points found in Step **S.8** are used to add instructions to the CFG that define the physical registers representing each predicate at a given program point. E.g., the predicate of a loop's first block is always true when initially entering the loop (assuming the path leading to the loop is enabled) and updated at the end of each iteration. When a block branches to two different alternatives, each target's predicate is set based on the result of the comparison instruction and whether the predicate of the original equivalence class is set (if a path is disabled, both of its resulting branching paths are also disabled.)

S.12 **Block Linearization & Merging**: The CFG blocks are sorted into a topological order and merged around loop boundaries. In Figure 6.1b, blocks b, c, d, and e become one block, g and h become another, and the rest stay the same. All branch instructions are deleted, and new branches are added to manage each loop's iteration. The topological order ensures that each block is executed before its successors in the original CFG, even without the explicit branching. The predicate

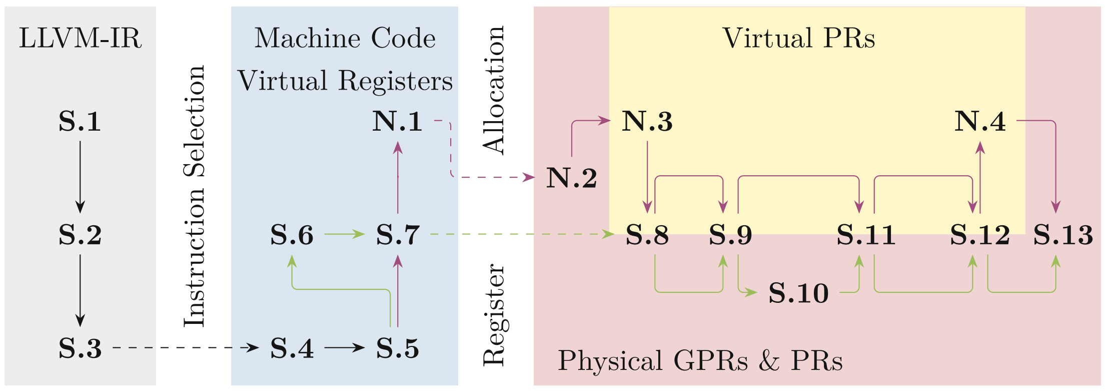


Figure 6.2: Single-path transformation compilation pipeline. Black transitions are shared, while the original transformation takes the green transitions, and the improved transformation takes the purple.

values are now what decides which code is enabled at run-time. All blocks within a loop are merged to avoid unnecessary branching, except around nested loops. We now have the single execution path like in Figure 6.1b but with the merged blocks.

S.13 **Instruction Scheduling**: At this point, the code still does not adhere to the timing requirements of Patmos, nor does it use an efficient order. E.g., the delay slot of a load is not respected—i.e., the result of the load is not available in the following cycles—meaning the use of the loaded value could come too early. The scheduler ensures that such timing requirements are managed and that the schedule is optimized by finding a better order for the instructions.

Steps **S.10** through **S.12** are what was described in [17] as *SPReduce*. Our steps describe the same implementation only in more detail.

This concludes the original single-path transformation. An overview of the steps is given in Figure 6.2 and follows the black and green transitions. What follows theses steps in the compiler are further steps to get the code ready for emission into the final binary assembly. These steps include further optimization, e.g., to eliminate any unneeded no-ops in delay slots, alignment corrections, and function splitting to ensure correct use of the special instruction cache Patmos uses [45]. These steps are not specific to single-path code and will not be further discussed.

6.5.4 Challenges

This section will discuss the challenges facing the original single-path transformation in terms of performance, maintainability, and incorrect code generation. The challenges resulted in the decision to reimplement parts of the transformation.

Performance

The LLVM project aims to make compiler implementation as easy as possible by providing as many generic facilities as possible that can be reused across languages and architectures. As such, it provides many *passes* that transform code in specific ways to achieve specific results. Steps **S.1** and **S.2** are such passes and are called for use in the single-path transformation. The generic register allocator is also provided and runs between Steps **S.7** and **S.8**. An important property of all LLVM-provided functionality is that it has been iterated on and optimized by hundreds or thousands of developers over many years. As such, any custom implementations of the same functionality must be well-optimized to outperform the generic functionality.

Several examples of sub-optimal code generation in the original transformation can be linked to not reusing LLVM's functionality. First, take the use of `r26`. As we have described, the lack of virtual registers forces `r26` to be used as a counter for all loops and for facilitating PR spilling/restoring. Since we do not have access to register allocation, we must ensure that each use of `r26` does not affect all other uses. If we need a loop counter, we cannot keep it in `r26` all the time, as a nested loop might need it, or a predicate might need to be spilled or restored. Therefore, the original single-path transformation is forced to preemptively spill loop counters to the stack at the start of each iteration and restore them at the end.

Another inefficiency regards the management of predicates. Because the transformation comes very late in the compilation process, we are forced to work directly with real instructions. This restricts our ability to use some of LLVM's optimization passes to make our predicate management more efficient. As an example, LLVM provides a *register coalescing* pass which can optimize unneeded movement of data between registers. However, it only works in earlier passes where all data movement is done using pseudo-instructions, called *copies*, and virtual registers. Coalescing might, e.g., notice that one virtual register always gets its value from the same source registers. It can then simply replace any use of the former register with the latter. The original transformation cannot do this, as it does not include the required analyses, and so always emits real data-movement instructions, even when not strictly necessary. The same kind of problem also arises around unneeded predicates. Not all equivalence classes end up including any predicated instructions. E.g., a block at the end of a loop might have its own class but includes only counter-management instructions, which are not predicated. As such, the register that is assigned that block's class is often unused. LLVM has a pass that can notice such patterns and eliminate the definitions of such registers. However, these passes are not run after the original transformation.

Lastly, insufficient development resources were available to implement the custom allocator. This is highly likely to result in worse allocation efficiency. The generic allocator, however, likely uses a much more mature algorithm and implementation, all but guaranteeing that it can produce better allocations if given the chance.

Maintenance

Another essential property of LLVM's generic functionality is that it is continuously maintained by the LLVM project and everyone who works on it. This means a reduced maintenance burden on architecture-specific additions to LLVM, likely resulting in fewer bugs and errors. The sub-optimal reimplementation of functionality for the single-path transformation is not only exemplified in the poor performance observed in Section 6.7 but also in the structure of the transformation and known bugs.

First, notice how the register allocation of PRs is split into two steps. First, Step **S.6** must analyze the CFG before LLVM's register allocation to determine the number of spill slots needed. These spill slots come from the need to spill individual PRs and the `s0` register prior to entering loops, as described in Step **S.10**. While Step **S.6** is not a complete register allocation step, its results must align with the eventual results of Step **S.10**, which is not trivial to guarantee—and is indeed not accomplished in the original transformation. The problem is that predicates outside a loop sometimes need to be accessed from inside a loop. This happens when we have a nested loop, and the inner loop `breaks` out of the outer loop. In such a case, the predicate enabling the body of the outer loop must be set to false so that both the inner and outer loops are disabled for the rest of both their iterations. Because the allocation of spill slots for `s0` is done in Step **S.6**, the allocator was not implemented with knowledge of exactly where each loop's `s0` is spilled. It just knows that it needs to be spilled. So when the predicate should be updated during the break, some other physical PR is updated erroneously. A fix for this requires a significant reorganization of the code between steps **S.6** and **S.10** to maintain the knowledge about `s0` spill slots and enable the allocator to use it in its allocation.

6.6 The Improved Transformation

To alleviate the challenges the old single-path transformation faces, especially regarding performance, the improved transformation aims to put the transformation steps earlier in the compilation pipeline where more LLVM functionality is available. Most importantly, the aim is to move Steps **S.8** through **S.12** before register allocation. This would enable the use of virtual PRs, more LLVM optimization passes like register coalescing, and, crucially, the use of LLVM's generic allocator for PR allocation.

6.6.1 LLVM-IR & Machine Code Preparation

The initial steps of the single-path transformation are unchanged for the new implementation, with Steps **S.1** through **S.5** being reused. Since we do not perform our own register allocation, we remove Step **S.6** completely so that Step **S.7** is what follows. Before the first round of register allocation, the following step is run:

N.1 Loop Counter Insertion: In the original transformation, Step **S.12** added code before and at the end of loops that used `r26` as the loop counter. In the improved transformation, we assign a virtual register as a counter for each loop.[2] Using the pseudo-root optimization, we may identify loops that have no iteration variance [125]. We classify these loops as *countless* here and omit the counter register since it is unneeded. Using a virtual register for the loop counters means more optimal use of the GPRs and no contention between nested loops for the use of `r26` and for PR spilling.

What follows is LLVM's regular register allocation passes. Similarly to the original transformation, the code does not use PRs except for comparisons and conditional branches. However, unlike the original transformation, we insert the bulk of the single-path transformation before LLVM moves on to function prologue and epilogue insertion.

6.6.2 Main Transformation

At this point, LLVM thinks it is done with register allocation. This is reflected in the metadata assigned to the function, basic blocks, and instructions. To be able to do another round of register allocation, we clean up the code so that LLVM reverts to thinking register allocation is needed. We do two preparatory steps before the main transformation:

N.2 PR Revirtualization: All uses of physical PRs are reverted to virtual registers, so we can exclusively use virtual registers for the single-path transformation.

N.3 Loop Counter Cleanup: Loops will iterate even if the called single-path function is disabled. This is problematic, as the counter registers will be used for the iteration. However, the calling code does not know that the caller will mess with registers because, in the original CFG, the function is never called. Therefore, we must ensure that disabled functions do not overwrite any registers, not even those used for loop counters. We do this by scanning the function and adding any loop counter registers to the prologue and epilogue. Likewise, loops skipped in the original CFG will still need registers for iterating in single-path code, which other paths might use. If so, we ensure the register is spilled/restored before/after the loop, such that it does not affect the use of that physical register in other paths.

We then run updated versions of Steps **S.8** through **S.12**, excluding Step **S.10**, as LLVM now manages PR allocation. These steps are slightly updated to work with virtual PRs instead of physical ones and to use LLVM's pseudo-instructions for data movement such that LLVM may optimize it in later passes like register coalescing. We lastly add the second register allocation step:

[2]In reality, multiple virtual registers because static single assignment form requires us never to overwrite a register.

N.4 **Forced Reallocation**: We reuse the LLVM register allocation infrastructure to force a rerun of the generic register allocator. This time, only PRs are using virtual registers, so the previous allocation of the GPRs is not affected. The generic allocator does not have dedicated support for spilling the single-bit predicates. Therefore, it uses two generic code sequences for it: To spill, it issues two stores predicated on the PR that needs spilling. The first store is only enabled when the predicate is true, storing the stack pointer, which is guaranteed not to be zero. The second store negates the predicate and stores the `r0` register, which is guaranteed to be zero. The result is that the spill slot becomes logically equivalent to the predicate value after both stores are executed. To restore the predicate, the spill slot is loaded into `r26` (reserved for this purpose) and moved into the relevant physical PR. The result is that each single-bit predicate uses a 32-bit spill slot and two instructions for each spill or restore. Since Patmos has delayed loads, even for stack loads, a PR restore uses at least three cycles. However, unrelated instructions could fill the delay to reduce it to two effective cycles. As discussed later, the spill/restore code might be optimized not to need `r26` and only be a single instruction.

Lastly, as with the original transformation, we run the single-path instruction scheduler after LLVM has inserted the prologue and epilogue.

Figure 6.2 also gives an overview of the new single-path transformation following the black and purple transitions. Notice that steps **S.8** through **S.12** (excluding **S.10**) in reality have two different implementations, one for each transformation. However, they perform the same task and are implemented very similarly, so they are instead shown to reside in both the red and yellow sections, representing that they work on both physical and virtual PRs, respectively.

6.6.3 Challenges

The improved single-path transformation comes with its own challenges and room for further improvement. However, we believe the challenges are less severe and the improvement opportunities more easily implementable.

First, we discussed in Section 6.5.4 how performing the transformation after register allocation was detrimental to performance and maintainability. We would have preferred to implement the single-path transformation such that it fully preceded any register allocation. However, we could not move the transformation completely before LLVM's regular register allocation because of the conflicting needs of GPR- and PR allocation. This means several performance and maintainability problems persist. E.g., we are still forced to split up the transformation by having loop counter insertion (Step **N.1**) before the first register allocation and the rest of the steps after. This complicates the implementation—and likely the maintainability—in favor of performance. It would have been simpler if the single-path transformation could be done before any registers were allocated. However, this is impossible when using a generic allocator.

Table 6.1: Representative instruction sequences for spilling and restoring individual predicate registers. `pS` is the register to be spilled/restored. `slot` is the stack spill slot, while `pos` is a spill position in a 32-bit slot. `rS` is a GPR used as a spill slot instead of the stack.

	Spill	Restore

Original	`lws r26=[slot]`
	`nop`
	`bcopy r26=r26, pos, pS`
	`sws [slot]=26`
		`lws r26=[slot]`
		`nop`
		`btest pS=r26, pos`

Improved	`sws ( pS)[slot]=rsp`
	`sws (!pS)[slot]=r0`
		`lws r26=[slot]`
		`nop`
		`mov pS=r26`

Optimal	`bcopy rS=rS, pos, pS`	`btest pS=rS, pos`

Not having access to virtual GPRs also affects the efficiency of PR spilling. As described in Step **N.4**, we need at least two instructions for both spilling and restoring each predicate bit, in addition to a 32-bit spill slot. This is more expensive than the old implementation, as it was able to use one 32-bit spill slot for 32 predicates. The optimal solution would instead be using GPRs as spill slots for PRs. This would allow using a single `bcopy` and `btest` for spilling and restoring to a GPR. The GPRs would, in turn, be spilled when needed as part of the regular register allocation. Table 6.1 highlights the difference between the original, improved, and optimal spilling and restoring codes. Unrelated instructions may populate the no-ops in the original spill and improved restore codes, though this is not guaranteed to happen. We can easily see that spilling and restoring are still expensive operations in the improved transformation compared to the fact that only a single bit is spilled. The ideal solution is for it to amortize to only needing a single instruction, as in the *optimal* row, using the `bcopy` and `btest` instructions that Patmos specifically has to support spilling/restoring [51]. Notice that using a GPR as a spill slot will result in higher GPR pressure, sometimes requiring additional GPR spilling. However, this is still cheaper than our current methods.

The improved transformation can likely be updated to use virtual GPRs and negate the above challenges. For simplicity, the implementation currently avoids mixing virtual and physical registers. However, it should be possible to do so. With the proper metadata management, it should be possible for the generic allocator to also allocate virtual GPRs in Step **N.4**. Time constraints did not allow for its implementation, but it should be possible and worth the time investment.

6.7 Evaluation

We use a subset of the TACLe benchmark suite to evaluate our work [50]. We include only those programs that successfully compile for both transformations and produce the correct

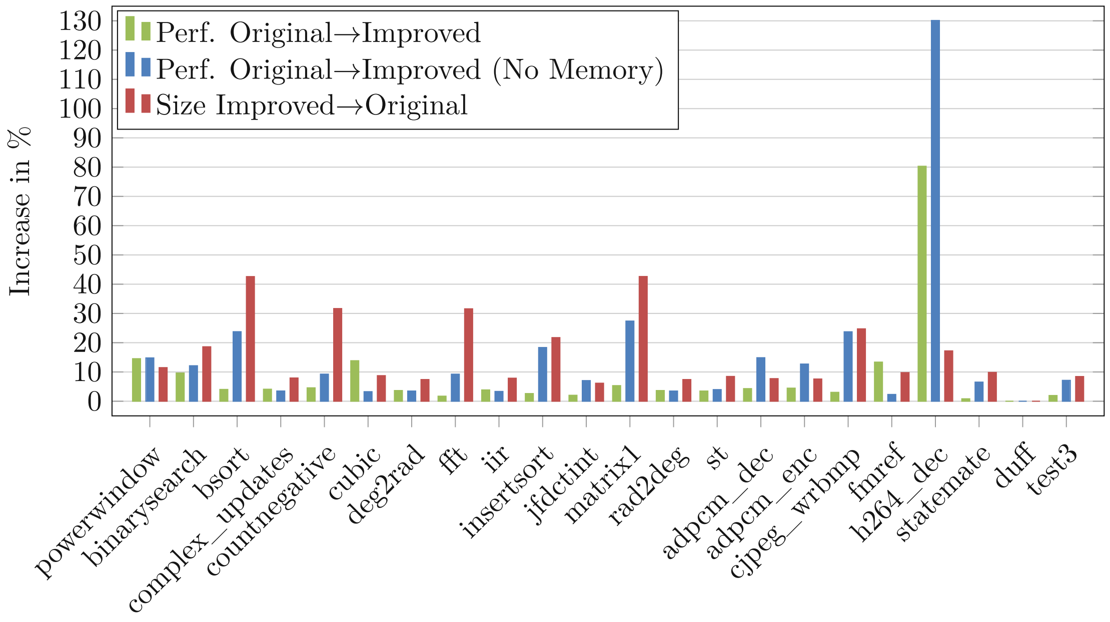


Figure 6.3: Performance and code size comparison between the original and improved single-path transformations.

result when run on Pasim, the Patmos ISA simulator.[3] We excluded programs for the following reasons: Eight programs had recursive calls, and three had loops without valid bounds. WCET-analyzable code can have neither unbounded recursion nor unbounded loops, so we disregard programs with it. The `fir2dim` program has a bug, meaning it never returns a successful exit code. Sixteen programs compiled wrong or threw an error during compilation with both the original and the improved transformation. Four other programs did so only when using the original transformation. Two programs, `pm` and `filterbank`, compiled and ran correctly but were so long-running that they saturated the simulator's cycle counter (32-bit). In the end, we have 22 programs to evaluate.

In Figure 6.3, we compare the performance increase of going from the original to the improved transformation. In the green bars, we see the increase based on raw execution time. While performance does increase for all but one benchmark program, the increase is mostly less than 10%. However, the `h264_dec` program sees an increase of 80 %. In Table 6.2, we see run-time statistics regarding stack cache and main memory accesses. In the first two columns, we can see that the number of bytes read from or written to the stack cache for `h264_dec` falls from 17 640 to 1 344 when switching to the improved transformation. Looking at the third and fourth columns, we see that the amount of time spent stalled waiting for main memory does not fall notably. This tells us that the improved transformation does not notably reduce the maximum amount of stack spill slots. Instead, only a large amount of unnecessary spilling is avoided. Looking at the

[3] All compilations use Clang's `O2` optimization level, the highest our implementations support.

Table 6.2: Run-time stack cache and main memory access statistics. For each column pair, the left column uses the original transformation, and the right uses the improved transformation.

	Stack Read/Write Bytes		Main Memory Stall Cycles	
powerwindow	3 376 590	1 217 072	19 981 710	17 455 788
binarysearch	62	16	504	462
bsort	81 422	16	840 441	840 294
complex_updates	109 284	91 480	1 206 975	1 158 339
countnegative	3 844	16	8 925	8 820
cubic	38 385 966	31 456 672	421 986 201	364 589 022
deg2rad	1 217 484	1 013 824	12 098 730	11 668 944
fft	23 523 016	144	551 750 976	551 750 703
iir	31 364	26 456	325 248	312 900
insertsort	962	96	16 989	16 884
jfdctint	224	72	6 846	6 762
matrix1	12 276	16	86 604	86 457
rad2deg	1 214 112	1 011 016	12 065 256	11 636 709
st	13 988 616	11 494 980	125 518 407	121 402 281
adpcm_dec	1 642	360	13 734	13 482
adpcm_enc	2 624	232	18 081	17 745
cjpeg_wrbmp	17 072	96	252 735	252 483
fmref	244 263 690	205 047 564	2 745 380 274	2 380 416 990
h264_dec	17 640	1 344	29 484	29 169
statemate	3 670	28	108 990	108 885
duff	16	16	2 562	2 562
test3	50 656 020	19 617 056	1 711 982 517	1 689 712 248

source code for `h264_dec`, we can see that it does a lot of looping, having nested looping that twice reaches five loops deep. Recall that for every loop iteration, the counter must be restored from its spill slot, decremented, and spilled back to the spill slot to ensure `r26` is ready for other uses; in this case, likely deeper nested-loop counters. This pattern of inefficiency around loop counters is most directly exhibited for `bsort`, where we see the stack use go from 81 422 bytes to 16. This reduction comes from having a loop that iterates 99 times nested within another loop iterating 99 times. Each iteration of the inner loop accesses the stack spill slot (4 bytes) twice to update the counter, resulting in $99 \times 99 \times 4 \times 2 = 78\,408$ accessed bytes. The outer loop also needs to access its counter every iteration, in addition to spilling and restoring `s0` before and after the inner loop. This gets us very close to the total accesses of the original transformation. In contrast, the improved transformation only uses the stack in the prologue and epilogue of the `main` function, proving that no spilling should be needed in the body.

For the blue bars of Figure 6.3, we have removed the stall cycles from main memory accesses from the execution time. Accessing memory takes up a large part of the execution time and can mask the effects of the improved transformation on the performance. Indeed, most programs see a larger increase in performance when filtering out main memory stalls, some to a significant degree. This is, again, especially notable for `h264_dec`, where the performance increases by 130 %. As we have already seen, stalls from memory accesses do not notably decrease when switching to the improved transformation. Removing the stall cycles—which account for 39 % of the improved transformation's execution time—highlights just how much time is wasted on spilling unnecessarily.

The difference between the green and blue bars is also notable in several cases where the performance increase is lower when filtering out main memory stall cycles. As an example, `cubic`'s increase goes from 14 % to 3.3 %. This unintuitive result stems from the difference in code size. The red bars show how many more instructions are in single-path code using the original transformation compared to the improved one. Often, the original transformation has significantly more instructions than the improved, with a maximum of 43 % for `bsort` and `matrix`. The lower amount of instructions in the improved transformation results in reduced execution time because the method cache needs to load fewer instructions from main memory. For `cubic`, code size is only 9 % higher for the original transformation. Moving instructions in and out of the method cache takes up 46 % of the execution time of the original transformation. The performance improvement of the improved transformation, therefore, almost all comes from the reduced code size, resulting in less waiting on the method cache. Therefore, not counting memory stalls results in the lower performance increase of the blue bars.

Lastly, notice how the `duff` program sees no difference between transformations for all our results. This is because it is so simple that the compiler optimizes it into a simple sequence of non-branching instructions long before the single-path transformation is reached. This means nothing needs to be done for the code to be single-path, and no difference between the transformations should appear.

Source Access

Patmos and its platform, T-CREST [15], are available as open-source and include the contributions of this paper. The Patmos homepage can be found at https://patmos.compute.dtu.dk/ and provides a link to the Patmos Reference Handbook [51], which includes build instructions.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-llvm-project. [4]

[4]Commit hash: 6460195495d7ca02d3001d84b1a9677328be317f.

6.8 Conclusion

Predicated instructions present a unique challenge that must be handled during register allocation. In this paper, we first discussed why the predicate registers cannot be allocated using a single run of a generic register allocator. We then presented how single-path code can use a generic register allocator to allocate PRs. The original single-path transformation used a custom allocator for the PRs, while our improved implementation instead leverages LLVM's generic register allocator to also allocate PRs by rerunning the allocation step after the improved single-path transformation is run. The improved implementation solves most of the challenges of the original but still has its challenges and room for further improvement.

Our results show that the new single-path transformation can significantly affect performance and code size, with best-case improvements of 80 % and 43 %, respectively. The performance improvement is primarily attributed to a much-improved allocation performance, with several cases of stack cache usage being reduced by several orders of magnitude.

CHAPTER 7

Predictable and Optimized Single-Path Code for Predicated Processors

Abstract—*Single-path code is a code generation technique for real-time systems that reduces execution time variability. However, doing so can incur significant execution-time overhead and does not guarantee constant execution times. In this paper, we address the performance challenges of single-path code and solve the variability issue. We present the repetition dominance relation to identify and optimize code blocks that are always executed a fixed number of times. We show that single-path code's instructions are uniquely easy to schedule, and we explore an extension to the Patmos architecture that allows additional instruction types in the second issue slot. Lastly, we present two techniques for ensuring that functions always perform the same number of accesses to memory, resulting in programs with constant execution time. We compare the performance of single-path code to that of statically analyzed traditional code. Our results show that single-path code's performance is mostly competitive while outright superior in several cases. However, pathological cases of poor performance are still observed.*

7.1 Introduction

For real-time systems, ensuring the timely execution of a program is often just as important as achieving the correct result. Execution times can exhibit substantial variations between different runs of a program. Multiple factors contribute to this variability, ranging from intrinsic aspects of the program's structure to the specific initial state of the hardware. The intricacies of contemporary systems make it exceedingly challenging to predict with precision the exact duration of a given program's execution. Establishing the worst-case execution time (WCET) is often a challenging part of a real-time system's development [2, 126, 127].

Single-path code generation focuses on removing execution time variability from programs [13]. By converting each function into a single sequence of instructions, input data cannot influence control flow and, thus, execution times. All program runs will exhibit the same execution time, meaning a simple measurement of the execution time using arbitrary inputs suffices to provide the WCET. Having constant execution times (CETs) provides multiple benefits; chiefly, the simplification of real-time system design but extends also to system safety [69].

Previous work using single-path code has had three major hurdles, which we address in this paper. The first hurdle is the propensity of single-path code to execute unneeded code. When no optimizations are performed, single-path code executes all paths through a program for every run, which is how execution time variability is eliminated. It also performs all function calls on all paths, even if a program path is unneeded on a given run. However, many paths and calls can be statically guaranteed not to have execution variability, causing single-path code to needlessly correct for variability that is absent, reducing performance. We present the *repetition dominance* relation for control-flow graphs; a variation of the well-known dominance relation that considers repeated visitations of nodes. We use this relation to identify functions and basic blocks that are always executed a fixed amount. These functions and blocks can then be optimized to reduce the amount of superfluous executions in single-path code.

The second hurdle is that of instruction scheduling. Single-path code has the unique characteristic of very few branching instructions and huge basic blocks with much instruction-level parallelism (ILP). This is ideal for multi-issue pipelines that can take advantage of the ILP. Existing single-path code schedulers can neither use single-path code's unique characteristics for efficient scheduling nor exploit multi-issue processor pipelines for additional performance. In this paper, we discuss how single-path code is uniquely easy to schedule and how control-flow information can be used for additional scheduling benefits. We present a heuristic list scheduler that efficiently schedules single-path code and can utilize the second issue slot of the Patmos processor for additional performance.

The last hurdle is that previous work on single-path code does not account for execution time variability induced by accessing memory. Accessing memory incurs a multi-cycle latency on even the simplest of platforms. If different paths through a program access memory in differing amounts, the cumulative latency from the accesses is also likely to differ. Single-path code using predication does not account for this since memory accesses are only effectual for the path actually needed in a given run. As such, execution times may vary even when single-path code eliminates input-dependent control flow. We address this shortcoming of single-path code to ensure programs have CETs. We present techniques that account for the variability in memory accesses and compensate for them by strategically performing compensatory accesses such that all executions result in the same number of accesses and, thus, the same execution time.

The contributions of this paper are: (1) A formalization of the repetition dominance relation and algorithm for calculating it, accompanied by proofs of correctness; (2) a description of optimizations for single-path code based on our relation; (3) a dual-issue scheduler that accounts for single-path code's unique characteristics; (4) a description of how to correct for memory access variability with two specific techniques; and (5) an implementation of our work in a compiler that produces single-path code.

This paper is an extension of the papers [69] and [125]. First, the repetition dominance relation in this paper is a variation of the constant-loop dominance relation presented in [125]. The previous relation was incorrect in some edge cases, resulting in false negatives and reduced performance. In this paper, we formalize a correct relation and prove its properties and the correctness of the algorithm for calculating it. We use a different name for the relation to highlight that they are technically different and to represent the new definition better. The work of [69] is presented in this paper mostly intact, with minor variations to account for the new relation. However, this paper uses a different single-path transformation that is significantly more performant [128]. Therefore, the final results vary considerably from those presented in [69]. The work on instruction scheduling has not been presented elsewhere.

This paper is organized into ten sections: The following section presents related work. Section 7.3 provides background information to support understanding of the rest of the paper. Section 7.4 introduces the repetition dominance relation, an algorithm for finding repetition dominators, proofs of correctness, and the pseudo-root optimization based on this relation. Section 7.5 discusses single-path code scheduling and how to take advantage of its unique characteristics. Section 7.6 presents how memory accesses result in execution-time variability and how to compensate for it. Section 7.7 describes how we have implemented our work in an LLVM-based compiler. Section 7.8 evaluates the performance impacts of our optimization and scheduling and compares CET single-path code's performance to that of analyzed traditional code. Section 7.9 discusses additional possible optimizations to single-path code. Section 7.10 concludes.

7.2 Related Work

Single-path code was introduced as a code generation technique specifically for real-time systems [13]. It can be automatically generated from any WCET analyzable source code. Previous work has looked into leveraging single-path code's ILP for increased performance [19]. In the middle of the single-path transformation, knowledge about control flow was used to easily find basic blocks that can be merged, such that each leverages one issue slot of a dual-issue pipeline. In contrast, this paper leverages that single-path code consists of a few huge basic blocks after the transformation is finished. These blocks can then be scheduled individually to great effect using existing techniques. Single-path code requires specific hardware support, which can be hard to come by. Therefore, work has been presented that can allow single-path code to run on otherwise unsupported processors through an instruction filter [20]. Single-path code is also used as a security-improving measure since its predictable timing makes side-channel attacks more challenging to execute [38, 39, 40, 37]. Because single-path code uses predicate registers, register allocation must be predicate-aware. To avoid implementing a fully-fledged predicate-aware allocator, generic allocators can be used in a two-step process for efficient allocation of both predicate- and general purpose registers [128]

The dominance relation is a fundamental concept in compiler construction used to identify the order in which various pieces of code are executed [129]. In the early days, a simple algorithm with a time complexity of $\mathcal{O}(n^4)$ was introduced to compute this relation [54]. Dominance was crucial for implementing, e.g., global common expression elimination and loop identification optimizations. It played a significant role in other important developments, including the efficient computation of static single assignment form [55], which, in turn, opened up opportunities for further optimizations [56, 57, 58]. Efforts have been made to reduce the runtime complexity of computing the dominance relation [59, 60, 61]. The current state-of-the-art offers an algorithm with a runtime of $\mathcal{O}(m\alpha(m, n))$, where n is the number of nodes, m is the number of edges, and α is the inverse Ackermann's function [62]. There have also been proposals aimed at finding a linear time algorithm [6, 5, 4]. However, the challenge lies in translating the theoretical runtime complexity into practical implementations that outperform older, non-linear algorithms.

Single-path code with CET can be an alternative to WCET analysis. The aiT static WCET analyzer has been used in industry to guarantee the runtime of safety-critical systems [126]. Many other academic and open-source tools are also used for researching various aspects of WCET analysis [130]. The challenge with static analysis is the increasing complexity of modern systems and the resources required to support each architecture and platform individually. The authors of [127] analyzed various static and measurement- or estimation-based analyses to assess their trustworthiness and practical applicability. They conclude that no technique is universally superior and all have some challenges. For example, fully static and formal analyses can be the most trustworthy but are also the least practical. They should, therefore, only be used for the most safety-critical parts of a system. The authors of [131] leverage the rigorousness of static approaches for worst-case path identification while, for practicality, using a language-processing-based WCET estimation. Bettonte, on the other hand, explores the potential of quantum computing for better static WCET analysis [132]. She concludes that while quantum computers can theoretically be used for the analyses, currently accessible machines have technical limitations that limit their applicability. It is also still an open question of identifying when a problem can be efficiently solved using quantum approaches.

Knowing loop bounds is a fundamental prerequisite for conducting WCET analysis. In this context, any annotation language must be able to define these bounds [63]. Nevertheless, manual annotations can often prove laborious for programmers to provide and may introduce imprecision and errors. Consequently, substantial efforts have been invested in developing automatic methods for identifying loop bounds [64, 65, 66]. However, no method can determine loop bounds for all scenarios. Therefore, researchers focus on identifying situations where automatic derivation is feasible. For instance, upper loop bounds can be inferred by assuming that a loop terminates and then systematically exploring the state-space of the variables influencing the loop exit condition [67]. Naturally, this approach necessitates accompanying proofs to establish the termination of loops. Some loops cannot easily be assigned an integer iteration bound. Non-orthogonal nested loops need automatic identification and correction methods to get tight WCET bounds [133]. In addition to traditional techniques, machine learning has also been explored as a means to discover loop bounds [68]. In this paper, we exclusively rely on annotated loop bounds. Nevertheless, our work is compatible with any method of establishing them.

Loop bounds can be helpful in applications other than WCET analysis. The verification of program transformations and optimizations must take loop-bound expressions into account [134]. Likewise, knowledge about loop bounds can significantly influence how to handle and optimize loops in parallel programming [135, 136]. Lastly, WCET analysis is a subset of bound analysis problems, which generally look at the resource use of programs; with execution time being one such resource. For all these problems, loop bounds are relevant for tight bounds on resource use [137].

Task scheduling and schedulability analysis are critical steps in real-time system design and require bounds on execution time. The field is well-studied with textbooks and surveys focusing on various aspects of scheduling [21, 22]. While WCET bounds are often used as execution time bounds, sometimes more is needed. For best-case response time analysis, best-case execution times are also needed [23]. However, with CET programs, the best and worst cases are identical, simplifying the system design. The simplicity of CET can also be leveraged to find optimal task schedules that are otherwise unavailable if execution times vary [30, 31]. However, other work cannot use CET. For example, the authors of [24] propose a scheduling algorithm to reclaim unused execution time (when tasks terminate earlier than their WCET) to increase schedule performance. This would have no effect when using CET, as tasks would always use all of their allotted time.

The logical execution time (LET) paradigm is similar to CET, except it centers around removing variability from the timing of program outputs; i.e., programs should always produce their output at a specific time after initialization [28]. This guarantee is stronger than our CET guarantee. However, the LET paradigm is compatible with our work, as it still requires bounded execution times (usually acquired with a WCET analyzer.)

Instruction scheduling is a crucial step in the compilation process that aims to find the best order for executing the instructions. Trace scheduling is a time-tested method that looks for the most important paths through a program and schedules the instructions to optimize their execution [87]. For very-long instruction word (VLIW) architectures, where operations can be bundled for parallel execution, superblock scheduling refines trace scheduling to exploit more ILP [138]. VLIWs often have increased code sizes because of instruction placement concerns. The authors of [139] use a backtracking instruction scheduler capable of simultaneously increasing performance and reducing code size by exploiting the characteristics of newer VLIW architectures. VLIW architectures have also been clustered to take even more advantage of ILP. Scheduling then becomes part of the larger problem of cluster assignment and inter-cluster communication, all of which need to be optimized simultaneously [140].

For architectures that also support predicated execution, hyperblock scheduling is an alternative to superblock scheduling that uses predicated instructions to merge multiple execution paths [108]. Predicate-aware scheduling (PAS) accounts for whether two predicates are *disjoint* (never simultaneously true) to allow them to be bundled even when using the same hardware resources [105]. We include PAS in this work. However, instead of creating boolean expressions to find disjoint predicates [99], we use a path-based approach to find the relations between predicates.

For real-time systems, it is also possible to customize the scheduling techniques to reduce WCET instead of average-case execution time [141, 73, 142]. A common way of doing that is finding the worst-case paths and optimizing them first, even if other paths might be impacted negatively. An alternative to the preceding techniques is to focus on individual basic blocks and schedule them optimally [143, 144]. In this work, we show that basic block scheduling can effectively schedule single-path code on a VLIW, predicated architecture. We show that heuristic list scheduling is a simple and effective way of scheduling the huge basic blocks of a single-path code. Future work should then look at using optimal scheduling for when peak performance is essential.

7.3 Background

This section provides a review of background topics needed for understanding the rest of the paper. First, we present the Patmos instruction-set architecture (ISA) and its unique features used in this work. Then, we introduce various definitions and custom notations used in the rest of the paper. Lastly, we describe how traditional code is converted into single-path code.

7.3.1 Patmos

The Patmos ISA was intentionally crafted for real-time systems, featuring a RISC-style VLIW ISA that focuses on ensuring time predictability and optimizing for a low WCET [16]. Every Patmos instruction is associated with one of eight boolean predicate registers. When the predicate evaluates to true, the instruction is considered *enabled* and executes normally. If the predicate evaluates to false, the instruction is *disabled*. It still executes in the same amount of time but does not interact with memory or alter any registers. Effectively, the instruction becomes a *no-op* operation. Each instruction also has a predicate negation flag. When the flag is set, the instruction is enabled when its predicate register evaluates to false and vice versa. For example, take the instruction `add (p1) r1 = r1, r2`. It is an addition instruction, predicated on the `p1` predicate register. To negate the predicate, an exclamation mark would be added (i.e., `!p1`).

Patmos includes an in-order, dual-issue pipeline designed to maximize throughput while maintaining time predictability. Up to two instructions can be *bundled* together, which makes the processor execute them in parallel, each in one *issue slot*. Not all instructions can be used in both of Patmos' issue slots. Loads, stores, multiplies, branches, calls, and returns are among the instructions not allowed in the second slot. Instruction scheduling must refrain from bundling these instructions using the second issue slot. Patmos also has long instructions, which take up two instruction words to support 32-bit constants. Those instructions use both issue slots and cannot be bundled.

Another notable aspect of Patmos is its utilization of split caches. Instead of using an instruction cache, Patmos has a method cache that lets the compiler determine when specific blocks of code, or entire functions, should be loaded into the cache [45]. Consequently, method-cache misses only occur at predefined moments, such as during function calls and returns. Additionally, Patmos divides the conventional data cache into two distinct caches: a stack cache and a data cache [46]. The stack cache is also under the compiler's control and serves the call stack and function-local data. Patmos has typed load and store instructions with dedicated instructions for data access via the data cache, stack cache, or direct access to main memory. To ensure predictability, Patmos incorporates instructions in the function prologue and epilogue for spilling and restoring stack-frame data. This design choice ensures that the subsequent use of this data within the function does not result in cache misses. Notably, instructions related to spilling and restoring registers do not trigger stack-cache misses and, therefore, do not contribute to variability. This architecture enables us to disregard any instructions targeting the stack, all while upholding the correctness of our techniques. As we will elaborate later, the remaining execution-time variability arises from the interaction between predicated instructions and data access through the data cache or the main memory.

7.3.2 Definitions

To aid in the proofs of the following sections, we will define the terms and mathematical notation used. A summary of the notation is provided in Table 7.1. First, we assume all control-flow graphs (CFGs) have a unique *entry node*, which we will denote as $\star$. It is usually associated with the first basic block executed in a function. We only consider CFGs that are connected, meaning all nodes are reachable from the entry.

$x \rightarrow y$ denotes the existence of a directed edge in a CFG from the node x to the node y. $x \rightsquigarrow y$ denotes the existence of a path in a CFG from the node x to the node y. There

Table 7.1: Summary of uncommon notation.

$\star$	CFG entry node
$x \to y$	Edge between two nodes
$x \rightsquigarrow y$	Path between two nodes
$x \rightsquigarrow_{\setminus z} y$	Path excluding specific nodes
$P \circ Q$	Sub-paths sharing first and last node
$P_{\#x}$	Number of visits in a path to a specific node
$P\langle\ \rangle$	A specific named path
$\odot h$	Set of nodes in a loop
$\otimes h$	Set of nodes outside a loop
$\circlearrowleft h$	Set of latch nodes in a loop
$\overleftarrow{h}$	Set of exit nodes in a loop
$\downarrow h$	Lower loop iteration bound
$\uparrow h$	Upper loop iteration bound
$\dagger h$	Constant loop
$\dot{\circlearrowleft} h$	Loop unilatch
$\diamond h$	Loop uni-exit
$x_\bullet$	LPV of specific node
$x \gg y$	Traditional dominance relation
$x \mathbin{\circ\!\!\gg} y$	Repetition dominance relation
$\to^h, \rightsquigarrow^h, \gg^h, \mathbin{\circ\!\!\gg}^h$	Relation only within a specific loop sub-graph

may be any number of nodes between x and y or none. A path may also be a single node ($x = y$) with no edges. The $\to$ and $\rightsquigarrow$ operators can also be used with variables denoting paths. E.g., $P \to x$ denotes the existence of a path equivalent to P, but with an additional edge to x. We will use lower-case letters for single nodes, while paths will use upper-case letter. We use equality between paths or edges and a variable to mean the given path exists, and the path the variable represents is equivalent to the given path. I.e., $P = x \rightsquigarrow y$ means there is a path between x and y, and P is such a path. $P \circ Q$ denotes the existence of a path comprised of two edges or sub-paths (or a combination) that share a last and first node, respectively. E.g., $(x \to y) \circ (y \rightsquigarrow z)$ is equivalent to $x \to y \rightsquigarrow z$. $x \rightsquigarrow_{\setminus z} y$ denotes the existence of a path from x to y that never visits a node z (or set of nodes) except for x and y themselves as the first and/or last node. E.g., $x \rightsquigarrow_{\setminus x} y$ can be the path $x \to z \to y$. $(x \rightsquigarrow y)_{\#z}$ denotes the number of visits to z on a path from x to y. Likewise, $P_{\#x}$ denotes the number of visits in the path P. E.g., $(x \to y \to x)_{\#x} = 2$. The notation $name\langle path\rangle$ is used to quantify and identify paths in the CFG. E.g., $\forall P\langle \star \rightsquigarrow x\rangle$ quantifies for all paths P from the entry to x. We can also constrain the named path in a quantifier. E.g., $\exists (P\langle \star \rightsquigarrow x\rangle \to y)\ P_{\#x} = 1$ is equivalent to $\exists P\langle \star \rightsquigarrow x\rangle\ P \to y \Rightarrow P_{\#x} = 1$.

In Definition 1, we restate the definition of the traditional dominance relation. In simple terms, a node x dominates a node y ($x \gg y$) if every path from the entry to y visits x

Definition 1 (Traditional Dominance [54])**.** *Given a CFG with nodes x and y:*

$$x \gg y \Leftrightarrow \forall P \langle \star \rightsquigarrow y \rangle \; P_{\#x} \geq 1$$

Definition 2 (Natural Loop)**.** *Given a CFG with a loop containing the nodes L:*

$$\exists h \in L \; \forall x \in (L \setminus h) \qquad h \gg x \wedge \neg (x \gg h) \wedge x \rightsquigarrow h$$

Definition 3 (Loop Iteration Bounds)**.** *Given a CFG with nodes N and a loop header h:*

$$\forall x \in \odot h \; \forall P \langle h \rightsquigarrow_{\setminus \otimes h} x \rangle \; \forall y \in N \qquad \star \rightsquigarrow P \rightsquigarrow y \Rightarrow 1 \leq \downarrow h \leq P_{\#h} \leq 2 \leq \uparrow h$$

[54]. We use traditional dominance to define the types of loops we consider. A loop is a strongly connected component, meaning all nodes are reachable from each other. Definition 2 specifies *natural loops*, which have a unique *header node* that dominates all other nodes in the loop. Notice that no other loop nodes dominate h, and all have a path to h, ensuring they also have a path to each other (at worst, by looping). We only consider natural loops and disallow any other loops.[1] We denote the set of nodes in the loop with header h as $\odot h$. $\otimes h$ denotes the set of nodes in the CFG not contained in the loop. A *back edge* enters the header from another node in the loop. The source node of a back edge is called a *latch*. $\circlearrowleft h$ denotes the set of latches in the loop with header h. An *exit edge* is an edge that connects a node in the loop to one outside of it. An *exit* is the source node of an exit edge. $\overleftarrow{h}$ denotes the set of exits in the loop with header h. If two loops have the same header, we treat them as the same loop. We refer to these "merged" natural loops as a single *loop* moving forward. All loops are either disjoint or nested within one another and identified by their headers. We say a node's header is the header of the innermost loop containing the node. For consistency, we also consider the entire CFG as a pseudo-loop with the entry node as the header and no exit nodes [17]. Therefore, when we reference a "loop" or a "loop header", that includes the CFG as a whole and the entry node as its header.

We define loop iteration bounds in Definition 3. The lower bound ($\downarrow h$) specifies the minimum number of times a path must visit the loop header before an exit edge may be taken. It cannot be lower than one, as that would mean the loop cannot be entered at all. The upper bound ($\Uparrow h$) specifies the maximum number of times a path must visit the loop header before an exit edge must be taken. It cannot be lower than two, as a loop that only iterates once will never take a back edge, thus never looping. Any loop with $\uparrow h = 1$ can be turned into a non-looping set of nodes and edges. By definition, we assume no path violates the upper or lower loop iteration bounds.

We need to use our definitions on loop sub-graphs for some of the following proofs. In

[1] It is possible to construct a CFG with a loop with no unique loop header. We do not consider such loops.

such cases, the loop's header will replace any reference in the original definitions to $\star$. E.g., when using $\gg$, we will use $\gg^h$ to express that the dominance relation is applied to the sub-graph of the loop with header h. This means that we consider paths starting from h instead of $\star$, and we ignore back edges to h and exit edges leaving the loop of h. Nested loops stay the same except if they have back edges to the outer loop header or exit edges that also exit the outer loop. As such, $x \gg^h y$ (x and y are in the loop) means all paths from the loop header h to y—without any back edge being taken—visit x. We do similarly for edges and paths. I.e., $x \rightsquigarrow^h y$ refers to paths in the sub-graph of h only and is assumed not to include any back or exit edges for the loop of h. It is equivalent to $x \rightsquigarrow_{\setminus(\otimes h \cup h)} y$.

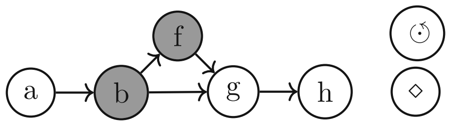
Each loop sub-graph also has two additional pseudo-nodes: The first is the *unilatch*, denoted as $\circlearrowleft h$ for a header h. It models the relationship between the latches and the header and has an incoming edge from each latch. The second is the *uni-exit* denoted as $\diamond h$, which models all the exits from the loop and has an incoming edge from each exit node.

Removing all back edges from the CFG results in an acyclic graph called a *forward control-flow graph* (FCFG) [17]. The FCFG is partitioned into *loop FCFGs* of the sub-graphs containing only nodes whose header is the loop header and the loop sub-graph unilatch and uni-exit [17]. For each loop, we, therefore, have a dedicated FCFG. Each node in the CFG is only in one FCFG, except the headers, which reside in the FCFG of their enclosing loop and in the FCFG of the loop for which they are headers. Exit edges are represented as edges from the header of the inner loop to the original target node in the outer loop. In Figure 7.1, we see the result of constructing the loop FCFGs of the CFG in Figure 7.2a. We can see in Figure 7.1a how the outer pseudo-loop contains a, g, and h, while we have highlighted b and f that represent their inner loops. Notice how in the original CFG, b does not have an edge to either f or g, but it does in the FCFG because of the exit edges $d \rightarrow f$ and $e \rightarrow g$. Notice also that the a-loop's unilatch and uni-exit are unconnected to the rest of the graph since there are no latches or exits in the pseudo-loop. Figure 7.1b is the FCFG for b's loop, which again contains b as the header. The unilatch gets edges from the two latches c and e, and the uni-exit gets edges from the exits d and e. Notice that FCFGs and loop sub-graphs are not identical. In loop sub-graphs, the inner loops are almost unchanged while their header in loop FCFGs replaces them.

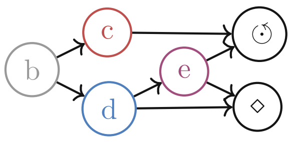
We use sub-graphs and FCFGs to complement the CFG during conversion to single-path code. Constructing the sub-graphs and FCFGs does not alter the CFG but makes reasoning about parts of it easier.

7.3.3 Single-Path Code

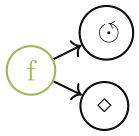
Single-path code uses predication to convert the branching control flow of a function into an instruction stream with only one path. All data-dependent execution-time variability is thereby removed from the instruction stream. We must use three techniques to convert



(a) Loop FCFG of loop a.

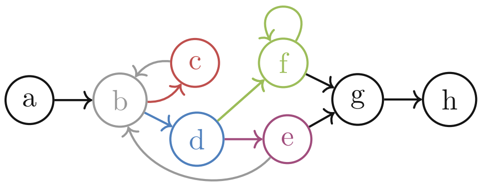


(b) Loop FCFG of loop b.

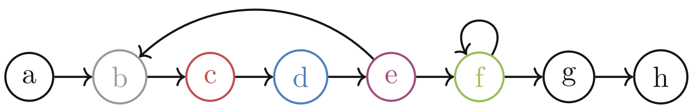


(c) Loop FCFG of loop f.

Figure 7.1: Loop FCFGs of the CFG in Figure 7.2a.



(a) Traditional



(b) Single-Path

Figure 7.2: Conversion of a program with branching control flow (top) to single-path (bottom).

a program into single-path code: If-conversion, loop conversion, and function conversion. To support the conversion, equivalence classes are used to manage predicates.

If-Conversion

First, we need to convert any conditional branching into a sequence of predicated instructions, such that only the needed path's instructions are enabled at runtime. The resulting code always executes all instructions in both paths. However, only one path's instructions are enabled at any given time. Looking at Figure 7.2, we can see the result of transforming a program into single-path code. The basic block b conditionally branches to c or d. The color coding of Figure 7.2a's blocks matches the conditions that led to that path being taken. In Figure 7.2b, the colors indicate that only if the corresponding condition is true will the block's instructions be enabled at runtime. As such, we can see how if-conversion results in b always leading to first c and then d. However, only if the red condition holds at runtime will c's instructions be enabled. The same holds for d, leading to either e or f in the traditional code but eventually leading to both in the single-path version. Notice how we have not colored the edges in the single-path version, as they are always taken.[2]

[2]Except the back edges, which are managed by loop conversion.

Loop Conversion

Loops may iterate a variable number of times depending on runtime conditions. Single-path code converts loops to always iterate the maximum possible number of times. Any superfluous iterations are disabled using predication. The result is invariable iteration while maintaining program semantics. The minimum and maximum iteration counts are usually given in the source code as an annotation on the loop. In our example, the program has two loops, one containing blocks b, c, d, and e, and the second containing only f. A single-path loop counts how many iterations are executed and keeps looping until the maximum is reached ($\uparrow b$ or $\uparrow f$). Inside the loop, the condition that traditionally breaks out of the loop is instead used as the predicate to all the instructions. This condition will become false at some point, meaning any further iterations will have disabled instructions.

Function Conversion

Lastly, single-path code also has to account for function calls. Say we have two branching paths, one of which performs a function call while the other does not. If we predicate the function call as we do for the rest of the instructions, it will not cause control to shift to the called function. Its instructions are, thus, not executed (neither enabled nor disabled) if the path it was called from is disabled. Instead, function conversion modifies all functions within single-path code to take an extra predicate register argument, specifying whether the function was called from an enabled or disabled path. The function body is then predicated on that register, meaning if it is called from a disabled function, it will be disabled as well, and vice versa. The call instruction in the caller is not predicated; instead, it is provided with the predicate of the calling code to pass on. This ensures all functions are always called and executed, regardless of whether their callers are enabled or disabled.

Equivalence Classes

Single-path code must define and maintain predicate registers throughout the program to emulate the original control flow. It does so by categorizing the basic blocks into groups that all must be enabled under the same circumstances. We call these groups *equivalence classes*. The *post-dominance* relation is the inverse of traditional dominance, where x post-dominates y if all paths that visit y must eventually visit x [145]. Given the nodes x, y, and z, and $y \rightarrow z$ and $z \rightsquigarrow x$. x is *control-dependent* on y if x post-dominates z but not y. x is also considered control dependent on $y \rightarrow z$. I.e., reaching z through $y \rightarrow z$ guarantees the execution of x eventually. Finally, we say two nodes are in the same equivalence class if they are control-dependent on the same set of edges.

Single-path code uses equivalence classes to track when blocks need to be enabled or disabled. It does so by identifying the equivalence classes of the blocks in each FCFG and assigning predicate registers to each equivalence class. The values of the predicate registers are updated at the source blocks of the dependence edges. The block's instructions are

Given a CFG with nodes N and loop headers H:

Definition 4 (Constant Loop)**.**

$$\forall h \in H \qquad (\downarrow h = \uparrow h) \Leftrightarrow \dagger\!\!h$$

Definition 5 (Last Possible Visit)**.**

$$\forall x, y \in N \ \forall P\langle \star \rightsquigarrow x \rightsquigarrow y\rangle \qquad x_\bullet \in P \Leftrightarrow \forall z, w \in N \ \forall Q\langle w \rightsquigarrow z\rangle \ (P \to Q \Rightarrow Q_{\#x} = 0)$$

Definition 6 (Repetition Dominance)**.**

$$\forall x, y, z \in N \qquad x \mathrel{\circ\!\!\gg} y \Leftrightarrow \exists v \in \mathbb{N} \ \forall (P\langle \star \rightsquigarrow y_\bullet\rangle \rightsquigarrow z) \ P_{\#x} = v$$

then predicated by the register assigned to their class. The aforementioned colors in Figure 7.2 match each equivalence class. First, note how a, g, and h are in the same equivalence class since they all depend on no edges (i.e., if the function is called, they will all eventually be executed.) The rest of the blocks are in their own equivalence classes since each has a unique set of control dependencies in their respective FCFGs. E.g., b is dependent on no edges as the header of its FCFG, c is dependent on $b \to c$, and d is dependent on $b \to d$.

7.4 Repetition Dominance

In this section, we will define our variation of dominance. However, before we can do that, we need to further specify the type of CFGs we use. We will then formally define our relation with accompanying theorems and proofs. Based on this, we will present an algorithm for calculating our relation and a proof for its correctness. Lastly, we will present an optimization that uses our relation to optimize single-path code.

7.4.1 Last Possible Visit

In Definition 4, we define a *constant loop* as having equal lower and upper loop iteration bounds. We consider CFGs with an optional label ($\dagger$) on the headers of loops, which signifies if a loop is constant. Therefore, we do not need to know the exact values of $\downarrow h$ and $\uparrow h$ to know if a loop is constant.

Definition 5 defines the *last possible visit* (LPV) of a node. A path contains the LPV of a node if the path cannot be extended to revisit the node. Note that the LPV of a node is only relevant in the presence of loops, as, without them, any visit is also the LPV. Intuitively, the LPV would be in the final iteration for constant loops.[3] E.g., the LPV of a header h ($h_\bullet$) would always be in the final iteration of a constant loop. For

[3]Not strictly true, see Lemma 1.

Lemma 1. *Given a CFG with a loop with header h:*

$$\forall x \in \odot h\ \forall y \in \otimes h\ \forall P\langle \star \rightsquigarrow Q\langle y \to h \rightsquigarrow_{\setminus \otimes h} x\rangle\rangle \qquad (\not\dagger h \wedge x_\bullet \in P) \Rightarrow Q_{\#h} \geq (\uparrow h - 1)$$

Proof. We show that assuming the negation results in a contradiction:

$$\begin{aligned}
&\exists x \in \odot h\ \exists y \in \otimes h\ \exists P\langle \star \rightsquigarrow Q\langle y \to h \rightsquigarrow_{\setminus \otimes h} x\rangle\rangle \qquad \dagger h \wedge x_\bullet \in P \wedge Q_{\#h} < (\uparrow h - 1)\\
&\Rightarrow^{x_\bullet \in P} \forall w, z \in N\ \forall (P \to R\langle w \rightsquigarrow z\rangle)\ R_{\#x} = 0\\
&\Rightarrow (w \in \odot h \Rightarrow^{Q_{\#h} < (\uparrow h - 1), Def.2} \exists T\langle P \to w \rightsquigarrow_{\setminus \otimes h} h \rightsquigarrow_{\setminus \otimes h} x\rangle\\
&\quad T_{\#x} = P_{\#x} + 1 \Rightarrow^{Def.5} x_\bullet \notin P \Rightarrow \bot) \Rightarrow w \notin \odot h\\
&\Rightarrow^{\not\dagger h, Q_{\#h} < \uparrow h} \neg(P \to R) \Rightarrow^{Def.2, Def.3} \exists k \in \odot h\ P \to k \Rightarrow \bot
\end{aligned}$$ □

Lemma 2. *Given a CFG with nodes N and a loop header h:*

$$\forall e \in \overleftarrow{h}\ \forall y \in \otimes h\ \forall P\langle \star \rightsquigarrow Q\langle y \to h \rightsquigarrow_{\setminus \otimes h} e\rangle\rangle \qquad (\not\dagger h \wedge e_\bullet \in P) \Rightarrow Q_{\#h} = \uparrow h$$

A proof similar to that of Lemma 1 can be used to show that this lemma holds.

Lemma 3. *Given a CFG with nodes N:*

$$\forall x \in N\ \forall P\langle \star \rightsquigarrow x\rangle\ \exists y \in N\ \exists Q\langle P \rightsquigarrow y\rangle \qquad x_\bullet \in Q$$

Proof. The negation to Lemma 3 is:

$$\begin{aligned}
&\exists x \in N\ \exists P\langle \star \rightsquigarrow x\rangle\ \forall y \in N\ \forall Q\langle P \rightsquigarrow y\rangle\ x_\bullet \notin Q\\
&\Rightarrow^{x_\bullet \notin Q} \neg(\forall w, z \in N\ \forall R\langle w \rightsquigarrow z\rangle\ (Q \to R) \Rightarrow (R_{\#x} = 0))\\
&\Rightarrow \exists w, z \in N\ \exists R\langle w \rightsquigarrow z\rangle\ (Q \to R) \wedge (R_{\#x} \neq 0)
\end{aligned}$$

Since all paths are finite, we can always find a y with no successors. In such a case, no R exists with an edge from Q. The negation of Lemma 3 is therefore false using a counter-example, meaning the lemma holds. □

nested loops, the LPV would be when all nested loops are in their final iteration. The LPV would always be in the last executed iteration of variable loops, so different paths with different numbers of loop iterations will have the LPV in different iterations. As a short-hand, we write $\exists P\langle \star \rightsquigarrow x_\bullet\rangle$ to mean $\exists P\langle \star \rightsquigarrow x\rangle\ x_\bullet \in P$ and likewise with $\forall$.

Lemma 1 states that if a node is in a constant loop, its LPV can only be in that loop's last or penultimate iteration. For example, look at the CFG in Figure 7.3 and assume the loop with header b is constant. b and c's LPVs are in the final iteration, while d's LPV is in the penultimate iteration because the exit must be taken from c in the final iteration, skipping d. LPVs cannot be in earlier iterations than the penultimate, as future iterations might still visit the node, contradicting Definition 5. Lemma 2 builds on this

$$\begin{tikzcd}
 & a \arrow[d] \\
 & b \arrow[d] \\
d \arrow[ur] & c \arrow[l] \arrow[d] \\
 & e
\end{tikzcd}$$

	$\gg$	$\circ\!\!\!\gg$ $\dagger b$	$\circ\!\!\!\gg$ $\neg(\dagger b)$
a	a, b, c, d, e	a, b, c, d, e	a, b, c, d, e
b	b, c, d, e	b, c, d, e	
c	c, d, e	b, c, d, e	
d	d	b, c, d, e	
e	e	e	e

Figure 7.3: Example CFG with the traditional and repetition dominance relations.

to show that LPVs of exit nodes must strictly be in the final iteration. Lastly, Lemma 3 shows that if a path reaches a given node, it is always possible to extend the path such that it includes the LPV of the node. Either the path already includes the LPV, or we can continue looping until we reach it.

7.4.2 Definition & Theorems

Definition 6 formalizes the repetition dominance relation. In simplified terms, a node x repetition-dominates a node y ($x \circ\!\!\!\gg y$) if every path from the entry to $y_\bullet$ visits x a fixed, non-zero number of times. In the rest of the paper, any unqualified mention of *dominance* refers to $\circ\!\!\!\gg$, while we use *traditional dominance* for $\gg$.

Looking at Figure 7.3, we can see the traditional and repetition dominance relations for the given CFG. Our relation is shown with the loop headed by b being both constant and variable. Notice how the nodes dominate each other in the constant loop when using our relation. Because there is only one LPV for each node, the others are always visited the same number of times before that. E.g., $d \circ\!\!\!\gg b$ because $b_\bullet$ is in the final iteration, which means d was visited once for each preceding iteration. The fact that d is not visited in the final iteration does not matter. When the loop is variable, none of them dominate anything, not even themselves. The number of visits to other nodes may change because each node can have an LPV in different iterations. If the loop only iterates three times, d will be visited twice before $b_\bullet$ in the third iteration. However, if it iterates four times, d will be visited three times before $b_\bullet$ in the fourth iteration.

Before presenting an algorithm for calculating repetition dominance, we will look at some interesting properties of the relation. First, as shown in Lemma 4, the traditional and repetition dominance relations are identical in the absence of loops. Lemma 5 states that a dominator of a loop node must be visited a fixed number of times between the loop's header and latches. If the dominator (x) is not in the loop, that number is trivially zero. However, if x is also in the loop, it makes sense that it needs to be visited a fixed number of times. Otherwise, two paths within one iteration would exist, visiting x differing amounts, contradicting x being a dominator. Lemma 6 shows that for a loop node to dominate, it must be in a constant loop. A corollary to Lemma 6 is that any node in

Lemma 4. *Given CFG with nodes N:*

$$\forall x, y, z, w \in N \quad \neg(z \rightsquigarrow w \rightarrow z) \Rightarrow (x \gg y \Leftrightarrow x \mathbin{\circ\!\!\gg} y)$$

Proof. We derive both directions of the equivalence:

$$\begin{aligned} x \gg y &\Rightarrow^{Def.1} \forall P\langle \star \rightsquigarrow y\rangle \; P_{\#x} \geq 1 \\ &\Rightarrow^{\neg(z \rightsquigarrow w \rightarrow z)} y_\bullet \in P \wedge P_{\#x} = 1 \\ &\Rightarrow \exists v \in \mathbb{N} \; P_{\#x} = v \Rightarrow^{v=1, Def.6} x \mathbin{\circ\!\!\gg} y \end{aligned}$$

$$\begin{aligned} x \mathbin{\circ\!\!\gg} y &\Rightarrow^{Def.6} \exists v \in \mathbb{N} \; \forall (P\langle \star \rightsquigarrow y\rangle \rightsquigarrow z) \; P_{\#x} = v \\ &\Rightarrow^{v \in \mathbb{N}} P_{\#x} \geq 1 \Rightarrow^{Def.1} x \gg y \end{aligned}$$

□

Lemma 5. *Given a CFG with nodes N and a loop header h:*

$$\forall x \in N \; \forall y \in \odot h \qquad x \mathbin{\circ\!\!\gg} y \Rightarrow \exists v \in \mathbb{Z} \; \forall l \in \circlearrowleft h \; \forall P\langle h \rightsquigarrow^h l\rangle \; P_{\#x} = v$$

Proof. We show that assuming the negation results in a contradiction:

$$\begin{aligned} &\exists x \in N \; \exists y \in \odot h \qquad x \mathbin{\circ\!\!\gg} y \wedge \forall v \in \mathbb{Z} \; \exists l \in \circlearrowleft h \; \exists P\langle h \rightsquigarrow^h l\rangle \; P_{\#x} \neq v \\ &\Rightarrow \exists l, k \in \circlearrowleft h \; \exists Q\langle h \rightsquigarrow^h l\rangle \; \exists Q'\langle h \rightsquigarrow^h k\rangle \; Q_{\#x} \neq Q'_{\#x} \\ &\Rightarrow^{2 \leq \uparrow h} \exists R\langle \star \rightsquigarrow h\rangle \; \exists R'\langle h \rightsquigarrow^h y\rangle \qquad R \rightarrow Q \rightarrow R' \wedge R \rightarrow Q' \rightarrow R' \\ &\Rightarrow^{Lem.3} T\langle R \rightarrow Q \rightarrow R''\langle R' \rightsquigarrow y_\bullet\rangle\rangle \wedge T'\langle R \rightarrow Q' \rightarrow R''\rangle \\ &\Rightarrow^{Q_{\#x} \neq Q'_{\#x}} T_{\#x} \neq T'_{\#x} \Rightarrow^{Def.6} \neg(x \mathbin{\circ\!\!\gg} y) \Rightarrow \bot \end{aligned}$$

□

Lemma 6. *Given a CFG with nodes N and a loop header h:*

$$\forall x \in \odot h \; \forall y \in N \qquad x \mathbin{\circ\!\!\gg} y \Rightarrow \dagger h$$

Proof. We show that assuming the negation results in a contradiction:

$$\begin{aligned} &\exists x \in \odot h \; \exists y \in N \; x \mathbin{\circ\!\!\gg} y \wedge \neg(\dagger h) \\ &\Rightarrow^{x \mathbin{\circ\!\!\gg} y} \exists v \in \mathbb{N} \; \forall z \in N \; \forall (P\langle \star \rightsquigarrow y_\bullet\rangle \rightsquigarrow z) \; P_{\#x} = v \\ &\Rightarrow^{Def.3} \exists w \in \odot h \; \exists Q\langle h \rightsquigarrow_{\setminus \otimes h} w\rangle \qquad P = \star \rightsquigarrow Q \rightsquigarrow y_\bullet \wedge \downarrow h \leq Q_{\#h} \leq \uparrow h \wedge Q_{\#x} \geq 1 \\ &\Rightarrow (Q_{\#h} = \uparrow h \Rightarrow^{2 \leq \uparrow h} \exists Q'\langle h \rightsquigarrow_{\setminus \otimes h} w\rangle \qquad Q = h \rightsquigarrow_{\setminus \otimes h \cup h} Q' \\ &\qquad \Rightarrow^{Lem.5} Q'_{\#x} = Q_{\#x} - 1 \\ &\qquad \Rightarrow \exists P'\langle \star \rightsquigarrow Q' \rightsquigarrow y_\bullet\rangle \; P'_{\#x} = P_{\#x} - 1 \\ &\qquad \Rightarrow^{Def.6} \neg(x \mathbin{\circ\!\!\gg} y) \Rightarrow \bot) \Rightarrow^{\neg(\dagger h)} Q_{\#h} < \uparrow h \\ &\Rightarrow^{Def.2, Def.3} \exists P''\langle \star \rightsquigarrow h \rightsquigarrow^h x \rightsquigarrow_{\setminus \otimes h \cup h} Q \rightsquigarrow y_\bullet\rangle \qquad P''_{\#x} = P_{\#x} + 1 \\ &\Rightarrow^{Def.6} \neg(x \mathbin{\circ\!\!\gg} y) \Rightarrow \bot \end{aligned}$$

□

a variable loop cannot repetition-dominate anything, as we have seen in our example. Lemma 7 shows that a loop node dominates another loop node if and only if it dominates the header. I.e., a node either dominates all loop nodes or none. Lemma 8 extends this to dominator nodes outside of loops.

The lemmas we have presented so far highlight the possibility of using a recursive algorithm to calculate repetition dominators. The base case of such an algorithm would be a CFG with no loops, in which case any traditional dominance algorithm would return the correct result for repetition dominance as well. The results of a recursive call can then be used in the outer loop to calculate how the inner loop nodes relate to the outer.

To illuminate the relationship between nodes in and outside loops, we can first look at Lemma 9, which shows that if a node x in a loop dominates another node y outside the loop, the loop header must have a path to y. Otherwise, there would be no way for x to be on a path reaching y. Lemma 10 builds on this, to state that for a node x in a loop to dominate a node y outside the loop, x must dominate the loop header. Otherwise, it would be possible to create paths with different numbers of visits to x before the header that continue to $y_\bullet$, contradicting $x \mathrel{\circ\!\!\!\gg} y$. Lemma 11 further refines the requirements on x by stating that there cannot be two exits with a path between them visiting x. If that would be the case, two paths could be created: The first exits from the first exit, while the second takes the path to x, then the other exit, and then to y. The second path would have visited x one more time than the first, again contradicting $x \mathrel{\circ\!\!\!\gg} y$. Lemmas 6, 9, 10, and 11 give us strong requirements around nodes that dominate nodes outside their loop. However, that is assuming we already know that $x \mathrel{\circ\!\!\!\gg} y$. To be useful, we need to show that these requirements are also sufficient to identify nodes in loops that dominate nodes outside the loops.

Theorem 1 defines how to identify dominators across loop boundaries. Take a constant loop with a path to y and containing x. If x dominates the unilatch in the loop sub-graph and no path visits x between exits leading to y, then x dominates y. We are now ready to present an algorithm that calculates the repetition dominance relation.

7.4.3 Algorithm

Finding traditional dominators on acyclic directed graphs (DAGs) is done by calculating the dominance for each node in topological order. Algorithm 7.1 shows how to do so (*DAGDom*). Using topological order ensures that the dominance of a node's predecessors has been established before getting their intersection to result in the dominance of the current node (and remembering to add self-dominance.) This can be seen on line 3 of the algorithm. Because of Lemma 4, we can build an algorithm for calculating repetition dominance based on *DAGDom*. Our algorithm will be recursively run on the FCFG of the loops, meaning we have a DAG for each step (the FCFG). The base case would be a sub-graph without any loops (the inner-most nested loop), where *DAGDom* would return the correct result. After the recursive calls, we translate the results to dominators for the outer loops using Theorem 1.

Lemma 7. *Given a CFG with a loop with header h:*

$$\forall x, y \in \odot h \qquad (x \mathrel{\circ\!\!\gg} h \Leftrightarrow x \mathrel{\circ\!\!\gg} y)$$

Proof. We prove both directions of the equivalence by showing that assuming the negation results in a contradiction. We start with the rightwards implication:

$$
\begin{aligned}
&\exists x, y \in \odot h \ (x \mathrel{\circ\!\!\gg} h \land \neg(x \mathrel{\circ\!\!\gg} y))\\
&\ \Rightarrow^{\neg(x \mathrel{\circ\!\!\gg} y)} \exists P\langle \star \rightsquigarrow y_\bullet\rangle \ \exists Q\langle \star \rightsquigarrow y_\bullet\rangle \qquad P_{\#x} \neq Q_{\#x} \lor P_{\#x} = Q_{\#x} = 0\\
&\ \Rightarrow (\exists R\langle y \rightsquigarrow_{\setminus(\otimes h \cup h)} h\rangle \qquad P'\langle P \circ R\rangle \land Q'\langle Q \circ R\rangle \land (P'_{\#x} \neq Q'_{\#x} \lor P'_{\#x} = Q'_{\#x} = 0)\\
&\qquad \Rightarrow^{Lem.1} h_\bullet \in P' \land h_\bullet \in Q' \Rightarrow^{Def.6} \neg(x \mathrel{\circ\!\!\gg} h) \Rightarrow \bot)\\
&\ \Rightarrow P = \star \rightsquigarrow h_\bullet \rightsquigarrow^{h} y_\bullet \land Q = \star \rightsquigarrow h_\bullet \rightsquigarrow^{h} y_\bullet\\
&\ \Rightarrow^{x \mathrel{\circ\!\!\gg} h} P_{\#x} = Q_{\#x} \neq 0\\
&\ \Rightarrow^{x \mathrel{\circ\!\!\gg} h} \exists P''\langle h \rightsquigarrow^{h} y\rangle \ \exists Q''\langle h \rightsquigarrow^{h} y\rangle \ \ P = \star \rightsquigarrow h_\bullet \circ P'' \land Q = \star \rightsquigarrow h_\bullet \circ Q'' \land P''_{\#x} \neq Q''_{\#x}\\
&\ \Rightarrow^{2 \leq \uparrow h} P = R\langle \star \rightsquigarrow h\rangle \rightsquigarrow_{\setminus(\otimes h \cup h)} h_\bullet \circ P''\\
&\ \Rightarrow^{Def.2} \exists R'\langle y \rightsquigarrow_{\setminus(\otimes h \cup h)} h\rangle \qquad T\langle R \circ P'' \circ R'\rangle \land T'\langle R \circ Q'' \circ R'\rangle \land h_\bullet \in T \land h_\bullet \in T'\\
&\ \Rightarrow^{P''_{\#x} \neq Q''_{\#x}} T_{\#x} \neq T'_{\#x} \Rightarrow^{Def.6} \neg(x \mathrel{\circ\!\!\gg} h) \Rightarrow \bot
\end{aligned}
$$

We omit the proof for the leftwards implication, which can be similarly shown to contradict Definition 6 when negated. □

Lemma 8. *Given a CFG with a loop with header h:*

$$\forall x \in N \ \forall y \in \odot h \qquad (x \mathrel{\circ\!\!\gg} h \Leftrightarrow x \mathrel{\circ\!\!\gg} y)$$

Proof. Lemma 7 proves the lemma for $x \in \odot h$. We, therefore, only need to do the same for $x \notin \odot h$. We prove both directions by deriving them:

$$
\begin{aligned}
x \mathrel{\circ\!\!\gg} h &\Rightarrow^{Def.6} \forall z \in N \ \exists v \in \mathbb{N} \ \forall P\langle Q\langle \star \rightsquigarrow h_\bullet\rangle \rightsquigarrow z\rangle \ Q_{\#x} = v\\
&\Rightarrow^{x \notin \odot h} \exists R\langle h \rightsquigarrow_{\setminus \otimes h} h\rangle P = \star \rightsquigarrow R \rightsquigarrow z \land R_{\#x} = 0\\
&\Rightarrow^{Lem.1} (P = R'\langle \star \rightsquigarrow y_\bullet\rangle \rightsquigarrow_{\setminus(\otimes h \cup h)} h_\bullet \rightsquigarrow z \land R'_{\#x} = v) \lor\\
&\quad\ (P = R''\langle Q \rightsquigarrow^{h} y_\bullet\rangle \rightsquigarrow z \land R''_{\#x} = v)\\
&\Rightarrow^{R'_{\#x} = R''_{\#x}} P = Q'\langle \star \rightsquigarrow y_\bullet\rangle \rightsquigarrow z \land Q'_{\#x} = v \Rightarrow^{Def.6} x \mathrel{\circ\!\!\gg} y
\end{aligned}
$$

$$
\begin{aligned}
x \mathrel{\circ\!\!\gg} y &\Rightarrow^{Def.6} \forall z \in N \ \exists v \in \mathbb{N} \ \forall (P\langle \star \rightsquigarrow y_\bullet\rangle \rightsquigarrow z) \ P_{\#x} = v\\
&\Rightarrow^{h \gg y} P = \star \rightsquigarrow h \rightsquigarrow^{h} y_\bullet\\
&\Rightarrow^{Lem.1} h_\bullet \in P \lor \forall Q\langle P \rightsquigarrow_{\setminus(\otimes h \cup h)} h\rangle \ h_\bullet \in Q\\
&\Rightarrow^{x \notin \odot h} P_{\#x} = Q_{\#x} \Rightarrow^{Def.6} x \mathrel{\circ\!\!\gg} h
\end{aligned}
$$

□

Lemma 9. *Given a CFG with nodes N and headers H:*

$$\forall h \in H\ \forall x \in \odot h\ \forall y \in \otimes h \qquad x \circ\!\!\gg y \Rightarrow h \rightsquigarrow y$$

Proof. We derive the implication:

$$\begin{aligned}
x \circ\!\!\gg y &\Rightarrow^{Def.6} \forall z \in N\ \exists v \in \mathbb{N}\ \forall (P\langle \star \rightsquigarrow y_\bullet\rangle \rightsquigarrow z)\ P_{\#x} = v \\
&\Rightarrow^{v \geq 1} P = \star \rightsquigarrow x \rightsquigarrow y_\bullet \\
&\Rightarrow^{h \gg x} P = \star \rightsquigarrow h \rightsquigarrow x \rightsquigarrow y_\bullet \\
&\Rightarrow h \rightsquigarrow y
\end{aligned}$$

□

Lemma 10. *Given a CFG with nodes N and a loop header h:*

$$\forall x \in \odot h\ \forall y \in \otimes h \qquad x \circ\!\!\gg y \Rightarrow x \circ\!\!\gg h$$

Proof. We show that assuming the negation results in a contradiction:

$$\begin{aligned}
&\exists x \in \odot h\ \exists y \in \otimes h\ x \circ\!\!\gg y \wedge \neg(x \circ\!\!\gg h) \\
&\Rightarrow^{\neg(x \circ\!\gg h)} \exists P\langle \star \rightsquigarrow h_\bullet\rangle\ \exists Q\langle \star \rightsquigarrow h_\bullet\rangle \qquad P_{\#x} \neq Q_{\#x} \vee P_{\#x} = Q_{\#x} = 0 \qquad (7.1) \\
&\Rightarrow^{x \circ\!\gg y, Lem.9} \exists R\langle h \rightsquigarrow_{\setminus h} y\rangle\ P \circ R \wedge Q \circ R \\
&\Rightarrow^{Lem.3} \exists R'\langle R \rightsquigarrow y\rangle\ \exists P'\langle P \circ R'\rangle\ \exists Q'\langle Q \circ R'\rangle \\
&\quad y_\bullet \in P' \wedge y_\bullet \in Q' \wedge (P'_{\#x} = P_{\#x} + R_{\#x}) \wedge (Q'_{\#x} = Q_{\#x} + R_{\#x}) \\
&\Rightarrow^{(7.1)} P'_{\#x} \neq Q'_{\#x} \vee P'_{\#x} = Q'_{\#x} = 0 \\
&\Rightarrow^{Def.6} \neg(x \circ\!\!\gg y) \Rightarrow \bot
\end{aligned}$$

□

Lemma 11. *Given a CFG with nodes N and a loop header h:*

$$\forall x \in \odot h\ \forall y \in \otimes h \qquad x \circ\!\!\gg y \Rightarrow \forall e, i \in \overleftarrow{h}\ ((e \neq x \wedge e, i \rightsquigarrow_{\setminus \odot h} y) \Rightarrow \neg(e \rightsquigarrow^h x \rightsquigarrow^h i))$$

Proof. We show that assuming the negation results in a contradiction:

$$\begin{aligned}
&\exists x \in \odot h\ \exists y \in \otimes h \qquad x \circ\!\!\gg y \wedge \exists e, i \in \overleftarrow{h}\ ((e \neq x \wedge e, i \rightsquigarrow_{\setminus \odot h} y) \wedge e \rightsquigarrow^h x \rightsquigarrow^h i) \\
&\Rightarrow^{\star \rightsquigarrow e, Lem.3} R\langle \star \rightsquigarrow e_\bullet\rangle \\
&\Rightarrow^{Lem.2} P\langle R \rightsquigarrow_{\setminus \odot h} y\rangle \\
&\Rightarrow^{e \rightsquigarrow^h x \rightsquigarrow^h i} R \rightsquigarrow^h x \rightsquigarrow^h i \rightsquigarrow_{\setminus \odot h} y \\
&\Rightarrow^{e \neq x, Lem.2} \exists Q\langle R \rightsquigarrow^h x \rightsquigarrow^h i_\bullet \rightsquigarrow_{\setminus \odot h} y\rangle\ P_{\#x} + 1 \leq Q_{\#x} \qquad (7.2) \\
&\Rightarrow^{Lem.3} \exists R'\langle y \rightsquigarrow y\rangle P'\langle P \circ R'\rangle \wedge Q'\langle Q \circ R'\rangle \wedge y_\bullet \in P' \wedge y_\bullet \in Q' \\
&\Rightarrow^{(7.2)} P'_{\#x} \neq Q'_{\#x} \\
&\Rightarrow^{Def.6} \neg(x \circ\gg y) \Rightarrow \bot
\end{aligned}$$

□

Theorem 1. *Given a CFG with nodes N and a loop header h:*

$$\begin{aligned}&\forall x \in \odot h\ \forall y \in \otimes h \quad (\quad x \circ\!\!\gg y \Leftrightarrow (\quad h \rightsquigarrow y \wedge \dagger\!h \wedge x \circ\!\!\gg^{h} \circlearrowleft h \wedge \\ &\quad (\forall e, i \in \overleftarrow{h}\ (e \neq x \wedge e, i \rightsquigarrow_{\setminus \odot h} y) \Rightarrow \neg(e \rightsquigarrow^{h} x \rightsquigarrow^{h} i)) \quad) \quad)\end{aligned}$$

Proof. We first show that the leftwards implication holds by showing that assuming the negation results in a contradiction:

$$\begin{aligned}
&\exists x \in \odot h\ \exists y \in \otimes h \quad \neg(x \circ\!\!\gg y) \wedge h \rightsquigarrow y \wedge \dagger\!h \wedge x \circ\!\!\gg^{h} \circlearrowleft h \wedge \\
&\quad (\forall e, i \in \overleftarrow{h}\ (e \neq x \wedge e, i \rightsquigarrow_{\setminus \odot h} y) \Rightarrow \neg(e \rightsquigarrow^{h} x \rightsquigarrow^{h} i)) && (7.3)\\
&\Rightarrow^{\neg(x \circ\!\!\gg y)} \exists P\langle \star \rightsquigarrow y_{\bullet}\rangle\ \exists Q\langle \star \rightsquigarrow y_{\bullet}\rangle \quad P_{\#x} \neq Q_{\#x} \vee P_{\#x} = Q_{\#x} = 0 && (7.4)\\
&\Rightarrow^{h \rightsquigarrow y, y \in \otimes h} \exists e', i' \in \overleftarrow{h}\\
&\quad (P = \star \rightsquigarrow P'\langle h \rightsquigarrow_{\setminus \otimes h} e'\rangle \rightsquigarrow y_{\bullet}) \wedge (Q = \star \rightsquigarrow Q'\langle h \rightsquigarrow_{\setminus \otimes h} i'\rangle \rightsquigarrow y_{\bullet}) && (7.5)\\
&\Rightarrow^{x \circ\!\!\gg^{h} \circlearrowleft h, x \in \odot h, \dagger\!h, 2 \leq \uparrow h} P'_{\#x} \geq 1 \wedge Q'_{\#x} \geq 1\\
&\Rightarrow^{(7.4)} P'_{\#x} \neq Q'_{\#x}\\
&\Rightarrow^{\dagger\!h, Lem.5} P' = h \rightsquigarrow_{\setminus \otimes h} P''\langle h \rightsquigarrow^{h} e'\rangle \wedge Q' = h \rightsquigarrow_{\setminus \otimes h} Q''\langle h \rightsquigarrow^{h} i'\rangle \wedge P''_{\#x} \neq Q''_{\#x} && (7.6)\\
&\Rightarrow (e' \rightsquigarrow^{h} x \rightsquigarrow^{h} i' \Rightarrow^{(7.3),(7.5)} e' = x \Rightarrow P''_{\#x} = Q''_{\#x} \Rightarrow^{(7.6)} \perp)\\
&\Rightarrow \neg(e' \rightsquigarrow^{h} x \rightsquigarrow^{h} i')\\
&\Rightarrow^{likewise} \neg(i' \rightsquigarrow^{h} x \rightsquigarrow^{h} e') && (7.7)\\
&\Rightarrow (e' \rightsquigarrow^{h}_{\setminus x} i'\\
&\quad \Rightarrow^{Lem.5} \forall l \in \circlearrowleft h\ \forall R\langle i' \rightsquigarrow^{h} l\rangle\ (P'' \rightsquigarrow^{h}_{\setminus x} R)_{\#x} = (Q'' \circ R)_{\#x}\\
&\quad \Rightarrow P''_{\#x} = Q''_{\#x} \Rightarrow^{(7.4)} \perp)\\
&\Rightarrow \neg(e' \rightsquigarrow^{h}_{\setminus x} i')\\
&\Rightarrow^{likewise} \neg(i' \rightsquigarrow^{h}_{\setminus x} e')\\
&\Rightarrow^{(7.6), Lem.5} \forall l, l' \in \circlearrowleft h\ \forall P'''\langle P'' \rightsquigarrow^{h} l\rangle\ \forall Q'''\langle Q'' \rightsquigarrow^{h} l'\rangle \quad P'''_{\#x} = Q'''_{\#x}\\
&\Rightarrow^{(7.6)} P'' \rightsquigarrow^{h} x \rightsquigarrow^{h} l \vee Q'' \rightsquigarrow^{h} x \rightsquigarrow^{h} l' && (7.8)\\
&\Rightarrow (P'' \rightsquigarrow^{h} x \rightsquigarrow^{h} l\\
&\quad \Rightarrow^{P'''_{\#x} = Q'''_{\#x}} Q''_{\#x} > P''_{\#x}\\
&\quad \Rightarrow^{P''_{\#x} \in \mathbb{Z}} Q''_{\#x} \geq 1\\
&\quad \Rightarrow h \rightsquigarrow^{h} x \rightsquigarrow^{h} i' \wedge e' \rightsquigarrow^{h} x\\
&\quad \Rightarrow e' \rightsquigarrow^{h} x \rightsquigarrow^{h} i' \Rightarrow^{(7.7)} \perp)\\
&\Rightarrow \neg(P'' \rightsquigarrow^{h} x \rightsquigarrow^{h} l)\\
&\Rightarrow^{(7.8)} Q'' \rightsquigarrow^{h} x \rightsquigarrow^{h} l' \Rightarrow^{likewise} \perp
\end{aligned}$$

Lemmas 6, 9, 10, and 11 together constitute the rightwards implication. Note that Lemma 10 implies $x \circ\!\!\gg y \Rightarrow x \circ\!\!\gg^{h} \circlearrowleft h$. □

Algorithm 7.1 Traditional Dominators for DAGs

```
    DAGDom(s):                                          ▷ Starting node as input
 1: $D \leftarrow \emptyset$                            ▷ Dominator set
 2: for $b \mid b \in topological\_sort(s)$ do
 3:     $D[b] \leftarrow b \cup \bigcap\{a \mid \forall a \rightarrow b\}$     ▷ Find dominators
 4: end for
 5: return $D$
```

Algorithm 7.2 Repetition Dominators

```
    RDom(s):                                            ▷ Starting node as input
 1: $H \leftarrow$ Inner loop headers
 2: $D \leftarrow \emptyset$                            ▷ Dominator set for nodes of $fcfg(s)$
 3: $ID \leftarrow \emptyset$                           ▷ Dominator sets for inner loop headers
 4: for $h$ in $H$ do                                   ▷ Analyze inner loops
 5:     $ID[h] \leftarrow RDom(h)[\circlearrowleft h]$
 6: end for
 7: for $b \mid b \in topological\_sort(fcfg(s))$ do
 8:     $B \leftarrow \bigcap\{D[a] \mid \forall(a, b) \in fcfg(s)\}$
 9:     $B' \leftarrow \{x \mid \forall x \in B \wedge (x \notin fcfg(a) \vee \forall e, i \in \overleftarrow{a}\ \neg(e \rightsquigarrow^a x \rightsquigarrow^a i))\}$
10:     $D[b] \leftarrow B' \cup \begin{cases} b & \text{if } b \notin H \\ b \cup ID[b] & \text{if } b \in H \wedge \nmid b \\ \emptyset & \text{otherwise} \end{cases}$
11: end for
12: for $h, v \mid \forall h \in H \wedge \forall v \in ID[h]$ do
13:     $D[v] \leftarrow D[h]$
14: end for
15: return $D$
```

Algorithm 7.2 calculates the repetition dominance relation. Given the header of a loop (or entry node for the whole CFG), it returns a map from the loop nodes to the set of nodes that dominate it. Note that it also returns the unilatch dominators. These are used for the recursion but can be safely ignored in the result of the whole CFG.

The algorithm starts by recursively calculating repetition dominators for all inner loops and saving the unilatch dominators in ID (lines 4-6). It then runs the main loop to calculate the dominators of the nodes in topological order (lines 7-11). Notice that we are using $fcfg(s)$ (line 7), meaning we only look at the fcfg nodes. For each node, we get the intersection of its FCFG-predecessors' dominators (line 8). The B-set might include nodes that dominate all b's predecessors but are between two exits of a preceding loop. Therefore, we exclude such nodes (line 9) to ensure we adhere to Lemma 11. Lastly, we add self-dominance; if b is itself a header, we also add the recursive result if it is constant. Note here that we have no proof that nodes that dominate the unilatch automatically dominate the header, as that is not true in the presence of outer loops. However, it will only be false if the outer loop is variable (Lemma 6) or there is a node in the outer loop that is not dominated (Lemma 7). The outer calls either do not assign dominators on line 13 because the loop is variable or by the node not dominating the outer loop unilatch and therefore not being included in the dominators of the outer header. $D[b]$, therefore, cumulatively adheres to Theorem 1.

After the algorithm's main loop, we have yet to assign the inner-loop nodes any dominators. From Lemma 8, we know that only nodes that dominate the header can dominate the rest of the nodes. So, we assign all the loop nodes their header's dominators. The algorithm concludes by returning D, which now includes the dominators for all nodes.

7.4.4 Proof of Correctness

For a CFG with nodes N, we need to prove the following statement:

$$\forall x, y \in N \qquad (x \mathrel{\circ\!\!\ggg} y \Leftrightarrow x \in RDom(\star)[y]) \tag{7.9}$$

Proof. We will use an induction proof based on the depth of loops (k). A CFG without loops has a depth of $k = 0$. With $k = 1$, the CFG may have loops but no nested loops. With $k = 2$, it may contain loops that, in turn, may contain a nested loop without any deeper nesting, and so on.

Base case. For $k = 0$, the CFG does not have any loops. Looking at Algorithm 7.2, we can see that when $H = \emptyset$, the algorithm becomes identical to Algorithm 7.1. It is easy to see that Algorithm 7.1 is correct for DAGs. Lemma 4 therefore shows that the base case holds.

Induction hypothesis. Given a CFG with maximum loop depth k, we assume Equation (7.9) holds.

Induction step. We now need to show that the two directions of Equation (7.9) hold for $k+1$ (i.e. deeper loops). We do so by showing that the negations result in contradictions. For the rightward direction, we start with:

$$\exists x, y \in N \qquad (x \mathrel{\circ\!\!\ggg} y \land x \notin RDom(\star)[y])$$

For $x \notin RDom(\star)[y]$ to hold, $D[b]$ on line 10 cannot include x when $b = y$. For that to happen, either $x \notin B$ on line 8 or it is excluded on line 9. Since $x \mathrel{\circ\!\!\ggg} y$, Lemma 11 says x cannot be in a nested loop between two exits. As such, it cannot be removed on line 9 and must be absent from B. This means a predecessor must exist where $x \notin D[a]$. a cannot be a loop containing x: For $x \notin D[a]$ to hold the induction hypothesis requires $\neg(x \mathrel{\circ\!\!\ggg} a)$, contradicting $x \mathrel{\circ\!\!\ggg} y$. So, x must be outside all loops, meaning there is a path to y in the FCFG without x, which also contradicts $x \mathrel{\circ\!\!\ggg} y$, meaning the rightward implication must hold.

For the leftwards direction, we start with the negation:

$$\begin{aligned}
&\exists x, y \in N \qquad \neg(x \mathrel{\circ\!\!\gg} y) \land x \in RDom(\star)[y] \\
&\Rightarrow^{\neg(x \mathrel{\circ\!\!\gg} y)} \exists P\langle \star \rightsquigarrow y_\bullet \rangle \; \exists Q\langle \star \rightsquigarrow y_\bullet \rangle \qquad P_{\#x} \neq Q_{\#x} \lor P_{\#x} = Q_{\#x} = 0
\end{aligned} \tag{7.10}$$

If the right side of Equation (7.10) were to hold, it would mean y has a predecessor that also has no visits to x before it. But since $x \in RDom(\star)[y]$, we know $x \in RDom(\star)[a]$ for all predecessors a. Because we know $\neg(x \mathrel{\circ\!\!\gg} y)$ and that the induction hypothesis ensures $x \in RDom(\star)[a] \Rightarrow x \mathrel{\circ\!\!\gg} a$, if y is a loop header, x cannot be in y's loop. x also cannot be outside all other loops since that would mean there is a path to y excluding x, which would mean a predecessor would also get $x \notin RDom(\star)[a]$. Thus, x must be in a loop preceding y. However, since x must be in that loop's header dominators to eventually reach y's dominators, $x \in RDom(\star)[h] \Rightarrow x \mathrel{\circ\!\!\gg} h$. For all $RDom(\star)[a]$ to include x, they all must have h as a predecessor, meaning all paths to y go through h first and contain a fixed number of non-zero visits to x (all in h's loop). As such, $x \mathrel{\circ\!\!\gg} y$, which contradicts $\neg(x \mathrel{\circ\!\!\gg} y)$. The leftward implication, therefore, holds, meaning Equation (7.9) holds. $\square$

7.4.5 Pseudo-Root Optimization

As described in Section 7.3, function conversion makes functions in single-path code take a predicate argument to enable or disable their bodies. However, that is most intuitively unnecessary for any single-path *root function*; a function that is itself single-path but called from a non-single-path function (e.g., the main function.) A root is guaranteed to be enabled, making the predicate argument unnecessary. The original single-path implementation recognized this and special-cased root functions not to need the predicate argument [17]. This reduces the number of instructions needed for predicate management, which results in reduced execution time.

The optimization of root functions can also be used for other functions. Any function we can guarantee is always called enabled a fixed amount can be optimized as if it were a root. Taking this further, we call a block *dominant* if it repetition-dominates all end blocks (blocks without successors). Any function called from a dominant block can also be optimized. We can do so because it means we know exactly how many times the function is called from that point. Function conversion, therefore, does not need to account for variations in call amounts (as there is no variation). The callee in cases like these is called a *pseudo-root* since its code generation can be identical to a root's. Any function called from a root or pseudo-root function in a dominant block is also a pseudo-root. For example, assume block c in the CFG of Figure 7.3 has a call to a function, and its loop is constant. Then, the called function would always be enabled in all iterations of the loop. If the call was in d instead, all calls except the one in the final iteration would be enabled. In both cases, the number of calls would be constant; either the same as the number of iterations (for d) or one less (for c). The called function would, therefore, be a pseudo-root. However, if the loop was not constant, the number of calls would change depending on the iteration count for each program run, and the

Theorem 2. *The runtime condition enabling or disabling a repetition-dominant block is input-data independent.*

Proof. First, notice that enabling or disabling a block in single-path code directly relates to whether the block is executed or skipped in the traditional code. Therefore, we focus on the traditional version of our CFG instead of the single-path one. We use an informal proof by contradiction: Say we have a repetition-dominant block (n) and an edge (e) (in the original, non-single-path CFG) that traditionally dominates n and is dependent on a condition that is determined at runtime (by input data). Because e traditionally dominates n, if e is taken, n will be visited eventually. In the case where n is not in a loop, it is easy to see that there must be an edge dependent on the negated condition ($\neg e$) that results in n not being visited before the end block. A path using this edge would visit n zero times, contradicting the definition of the repetition dominance relation. The other case is where n is in a loop. If $\neg e$ were in the loop with n, it would mean n does not dominate the latches, which would allow a path to skip it in some iterations of the loop, meaning not all paths visit it the same number of times, a contradiction with the relation. If $\neg e$ were in an outer loop, the whole loop could be skipped using $\neg e$, similarly allowing a path to skip n, resulting in another contradiction. Lastly, if $\neg e$ were in an inner loop, the edge must lead to either a latch or exit in n's loop. If it leads to a latch, n cannot dominate the latch and so can be skipped on some iterations, resulting in a contradiction. If it leads to an exit, the number of iterations the loop takes is now input-data dependent (since $\neg e$'s condition is), meaning the loop is not constant, which is a contradiction to n being dominant (as it must then be in constant loops). □

called function would not be a pseudo-root. Another example of non-pseudo-roots would be any function called from the blocks c, d, e, or f in Figure 7.2a.

The pseudo-root optimization uses repetition dominance to explore the call tree from the root function(s) and identifies all pseudo-root functions. The single-path transformation then uses the information to optimize all pseudo-roots not to take a predicate argument. It also changes all calls to pseudo-roots to be predicated, so the functions are not called when a block is disabled. Lastly, constant loops that are dominant (i.e., their headers are dominant) are guaranteed always to be enabled. Therefore, we can avoid using a counter to ensure they always iterate the same amount. This optimization finds all such loops in pseudo-roots and omits adding code that counts iterations.

Note that a function may be called from both a repetition-dominant block and one that is not. For example, it could be called both in the entry block of a program and within only one path of a branch. In such cases, functions are duplicated, such that two versions are used: one that takes an additional argument and one that does not.

To use this optimization, we must be sure that it maintains single-path code's guarantee of executing the same instruction stream regardless of input data. We must show that not calling pseudo-root functions when the calling block is disabled maintains this guarantee.

Theorem 2 shows that repetition-dominant blocks are enabled and disabled in an input-data-independent manner. Therefore, they must be enabled and disabled in the same sequence in every function run. As such, the instruction stream that results from this optimization is identical to the old, non-optimized instruction stream, except with the instructions from specific calls removed. Since the code that is removed is always removed from the same place in the original stream, the resulting stream must also always be the same. Thus, our optimization maintains the single-path guarantee and is safe to use in single-path code.

7.5 Scheduling Single-Path Code

Instruction scheduling reorders the execution of instructions to optimize for some metric—performance, in our case. The instruction selection stage of a compiler produces machine-code instructions, or machine-code-like instructions, in some order that is known to be valid. The order is also often only in single-issue, meaning no bundling of instructions is present, wasting the resources of VLIW architectures. Instruction scheduling must then take the initial sequence and find a new one that better utilizes a processor's capabilities. Instruction scheduling is NP-complete, meaning it is impractical to aim for optimal solutions [146]. Instead, heuristic and greedy approaches are often used to produce acceptable schedules in manageable times.

This section discusses the unique characteristics of single-path code that should be accounted for when scheduling its instructions. We will first look at how to reduce instruction dependencies using equivalence classes, followed by how the control flow of single-path code affects scheduling techniques. Lastly, we discuss how equivalence classes can be used to increase ISA and hardware flexibility without further complicating the ISA design or hardware implementations.

7.5.1 Disjoint Equivalence Classes

Single-path code has big basic blocks. If we look at Figure 7.2b, we can see that merging the b-loop into one basic block would not affect the semantics of the program. Single-path code does precisely this to avoid executing branches between the blocks. The result is one big block comprised of the four original blocks. Repeating this for the rest of the CFG, we end up with only four merged blocks $\{a\}$, $\{b, c, d, e\}$, $\{f\}$, and $\{g, h\}$. Merged blocks will often have high ILP, as some of their constituent blocks are enabled while others are disabled. E.g., if block c is enabled, blocks d and e must be disabled. Such blocks' instructions can never affect each other and may thus be freely reordered without accounting for their register or memory accesses.

Whether instructions from different original blocks can be simultaneously enabled depends on their equivalence classes. If two predicates cannot evaluate to true simultaneously, they are called *disjoint* [99]. We use the same terminology for equivalence classes and instructions. In our example, the black class is not disjoint from all other classes, as it is always enabled. Likewise, the gray class, as the loop header, is also not disjoint from all other classes in the loop (red, blue, purple) since it must be enabled for any of the others to become enabled in the same iteration. The red and blue classes are disjoint since they are two sides of a branch, while the blue and purple classes are not disjoint since the blue must be enabled for the purple to be as well. In general, two classes are not disjoint if there is a path without back edges from the blocks of one class to those of the other.

For two instructions to be disjoint, their equivalence classes must be disjoint. However, there are additional requirements: 1) If two instructions are part of the same equivalence class, they are disjoint if they use the opposite negations of the predicate. In such cases, exactly one of them will always be enabled. 2) If their classes are disjoint, but either one or both instructions negate their predicate, then the instructions are not disjoint. The class disjointness only guarantees that their predicates are never enabled simultaneously but can be disabled simultaneously. Negating either one or both predicates nullifies this guarantee.

The disjoint relationships between instructions are used in instruction scheduling to ignore traditional scheduling dependencies between instructions in merged blocks, achieving PAS [105]. Assume block d contains the instruction `add (p1) r1 = r1, r2` and the block e contains the instruction `sub (p2) r1 = r1, r2`. Without knowledge of equivalence classes, there is no guarantee that `p1` and `p2` are disjoint. Therefore, the instructions' scheduling dependencies are as follows: They both read from and write to `r1`, meaning they have a true dependency (read must follow write), an anti-dependency (write must follow reads), and an output dependency (write must follow write.) The two instructions must maintain their order in the final schedule, and they cannot be bundled together to take advantage of ILP. In contrast, we can ignore all their dependencies when PAS is used. We can both reorder them, which might find free spots in earlier parts of the schedule, or they can be bundled together. Both options reduce the execution time.

7.5.2 Control Flow

Single-path code includes very little and exceedingly predictable control flow: If-conversion removes any branches unrelated to loops and loop conversion results in loops that always iterate a fixed amount. The result is that code is minimally affected by branching, meaning scheduling can more easily ignore branches without heavy performance penalties.

Single-path code also executes all paths. More accurately, its execution time is affected by all paths in the original CFG. Thus, traditional scheduling techniques may perform poorly when executing single-path code. For example, trace scheduling looks for the most executed path and schedules it optimally, even to the detriment of other paths [87]. Hyperblock scheduling for WCET likewise finds the worst-case path and aims to reduce its execution time even to the detriment of other paths' execution times [108]. That is not a problem for traditional code, as the WCET is still reduced. However, for single-path code, reducing the execution time of one path may cumulatively increase the execution time of other paths by more than the reduction, increasing overall execution time. For example, the worst-case path might be reduced by two cycles at the cost of increasing other paths by one each. If there are more than two other paths, single-path code's execution time (also the WCET) would rise instead of fall. Therefore, we must be careful when using traditional scheduling techniques on single-path code.

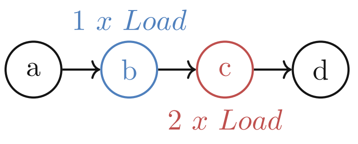
The control flow characteristics of single-path code and its big blocks mean simple scheduling techniques should produce good results. To show this, we will only use local scheduling of individual basic blocks without looking at their interplay with other blocks. We will use a simple heuristic list scheduler on the blocks to show that it can produce good results.

7.5.3 Permissive Dual-Issue

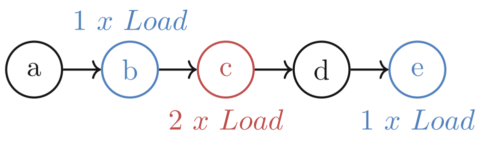
As mentioned earlier, the Patmos processor has a dual-issue pipeline that limits some instructions to only being allowed in the first issue slot. This limitation must be encoded into the instruction scheduler so that it does not schedule an instruction in the second issue slot that is not allowed to be there. That limits both the flexibility of the scheduler and resource utilization.

There are multiple reasons for limiting the use of the second issue slot. One reason is that of hardware complexity. If two memory access instructions are allowed to be bundled, the hardware must either have two ports that can access memory in parallel or seamlessly interleave the accesses in the background using one port. Both solutions require additional memory access circuitry. Another reason relates to the architectural state. The Patmos multiply instruction—like in many other architectures—implicitly outputs its result in the `sl` and `sh` registers, corresponding to the low and high halves of the 64-bit result, respectively. If two multiplies were bundled, the Patmos ISA would need two more registers for the second multiply to output to, increasing architectural state and complexity. A third reason is ambiguous control flow. If two branches or calls were to be enabled in the same bundle, where should execution continue? Simply disallowing some instructions like memory accesses, multiplies, and branches in the second issue slot simplifies both the ISA design and implementation.

PAS opens up the possibility of an alternative solution. Given that disabled instructions do not update registers or access memory, if we can guarantee that only one of each problematic instruction type may be enabled at a time, no complex ISA changes or hardware implementations are needed to support those instructions in the second issue slot. Suppose we can guarantee that only one of a bundled pair of memory access instructions is enabled. Only one port is needed in that case, and no interleaving is required. Likewise, with a set of multiplies, we would not need a second set of output registers, as, at most, one result is produced at a time. Bundled control-flow instructions are also not ambiguous if only one will ever be enabled at a time. We can make this guarantee using PAS. Any time we see two disjoint memory accesses, multiplies, or branches, we allow them to be bundled. The Patmos ISA currently prohibits that. However, as we will describe in



(a) Varying access counts



(b) Fixed access counts

Figure 7.4: A single-path CFG with varying memory accesses (left) can be corrected to issue a fixed number of accesses (right).

Section 7.7, we explore the impacts of this by implementing it in the Patmos simulator, Pasim.

7.6 Achieving Constant Execution Time

As we mentioned in Section 7.3, even though single-path code has a single instruction stream, it does not mean execution times are constant. Any system with multi-cycle memory access latency will exhibit varying execution times for single-path code, assuming that the given processor does not prompt the memory system on disabled memory accessing instructions. T-CREST is such a system. The Patmos ISA specifies a one-clock-cycle latency for loads (meaning there is a single-cycle load use delay). However, our standard FPGA-based platform has a larger than one clock cycle main memory access latency. Although, that access time is constant.

This section will describe why variability in memory accesses causes varying execution times. We will then describe the principles that can compensate for memory access variability. Lastly, we present two specific compensation techniques we implemented in our compiler: Opposite-predicate compensation (OPC) and decrementing-counter compensation (DCC), before briefly discussing the general requirements for using our work.

7.6.1 Memory Access Variability

Take the single-path program in Figure 7.4a: It branches at a to either b or c and converges again on d. The code in b issues a single load instruction, while c issues two load instructions. In its single-path form, only one of b or c's instructions are enabled simultaneously. While this does not result in variability for most instructions—since their execution time is identical whether enabled or disabled—this is not the case for instructions accessing memory. For example, enabled loads must query the memory system for the data. However, when disabled, no querying is done. While the load still takes time to run through the pipeline, no stalls are issued waiting for the memory system to return data. Thus, enabled loads (and other memory access instructions) incur a higher latency than disabled ones. In the default implementation of T-CREST on the DE2-115 FPGA board, this latency is 21 cycles.

Execution time variability is introduced if different paths through the program issue different numbers of memory accesses. In our example program, *b* only issues one load while *c* issues two. In a run of the program where *b* is enabled, memory access latency is incurred only once. When *c* is enabled, the latency is incurred twice. The difference in execution time is, therefore, 21 cycles on the T-CREST platform, depending on which path is enabled. Had the difference in the number of loads been bigger, the execution time difference would be proportional to it; for every additional load, the difference increases by 21.

While our example uses only direct access to the main memory, the variability is also exhibited in hierarchical memory systems. Caches or scratchpads often incur multi-cycle latencies. For example, accessing the shared scratchpad on T-CREST can incur a 13-cycle latency [48]. With caches, we also have additional latency from cache misses, compounding the variability.

7.6.2 Variability Compensation

To achieve CET, we must compensate for the uneven number of memory accesses in different program paths. That can be done in many ways. However, the general idea is to add instructions to the program that add execution time where needed. For example, this could be done by adding a set of predicated no-op instructions that are only enabled when their respective memory access is disabled. The downside would be having to match the number of no-ops to the specific latency of a given platform.

The general strategy we will use is the insertion of superfluous memory accesses, i.e., load and store instructions. We thus compensate for a given disabled memory instruction by executing another—enabled—compensatory memory instruction. In the general case, we would have to match each type of memory access instruction to the same type of compensatory instruction, i.e., loads must compensate for loads and stores for stores. However, loads and stores have the same latency on the T-CREST platform, letting us use loads to compensate for stores, too. Therefore, we will not distinguish between loads and stores in the rest of the paper.

All compensations will be done by loads using the `r0` register as both the source and destination. We can do that since address 0 is always valid to load from, and `r0` is a read-only register, meaning no data is changed in memory or registers by the compensatory loads. Figure 7.4b shows a possible compensation for the original program in Figure 7.4a. *e* is added with the same predicate as *b* and has one compensatory load. The result is that both paths through the program (initially either through *b* or *c*) now have two loads and, therefore, the same execution time.

We will now present two specific compensation techniques we implemented in our compiler. They perform differently depending on how much variability is in a given function. Both ensure the program performs the same number of memory accesses regardless of what happens at runtime.

Opposite-Predicate Compensation

The first variability compensation technique we present is designed to be simple to implement and have minimal overhead for low-variability code. OPC goes through all the instructions in a function. Whenever it encounters a memory access, it adds a compensatory load with the same but negated predicate. The result is that either the original instruction or the compensatory load is always enabled, meaning the memory access latency is always incurred once for each of the original memory instructions.

While this technique is easy to implement, it can also add significant overhead. Memory access latency is incurred for every memory instruction on all paths. It also adds many new instructions to the program, one new load for every existing memory instruction, meaning additional cycles are wasted on top of all the extra latency incurred. However, if a function has few memory-accessing instructions, this technique's overhead should be negligible.

Decrementing-Counter Compensation

The following variability compensation technique is designed to amortize the compensation overhead; i.e. the number of instruction needed to perform the compensation. The key is remembering that to achieve CET, we need all paths through a function to have the same number of enabled memory accesses. The number of accesses we need equals the number of accesses the path with the most accesses has. By analyzing the function, we can count how many accesses are on each path and find the path with the most accesses. We will designate its number of accesses as A_{max}. At the end of the function, a loop can perform extra accesses, depending on which path was taken, such that the final number of enabled accesses becomes constant (A_{max}).[4] Using a loop will minimize the number of instructions added to the function compared to OPC, which adds a compensatory load instruction for every ordinary memory-accessing instruction in the loop.[5]

Throughout the function at runtime, we will maintain a counter, c, which at the end of the function will contain the number of additional memory accesses we need to perform to reach A_{max}. c is initialized to A_{max} at the start of the function and decremented throughout. We use the edges between blocks to track where decrements must be inserted. Initially, we label each edge with the number of accesses its source block performs. To account for the accesses in the final block, we add its access count to all incoming edges. At this point, if we add a decrement in each source block matching the labels, we would have a working compensation. However, minimizing the number of added decrement instructions through constant propagation optimization will maximize performance. This compensation is only based on the control flow and does not depend on the single-path transformation. Additionally, the technique requires us to use a register to hold the counter, which lives throughout the function. Therefore, it is best to add the compensation before register allocation and let the single-path transformation run

[4]This loop must also be loop converted.

[5]Assuming there are many memory-accessing instructions in the function.

afterward. This differs from OPC, which needs to know which predicates are assigned to each instruction by the single-path transformation, forcing it to be performed afterward.

The two compensation techniques we have presented here are not the only possibilities. Our general strategy is flexible enough to accommodate others in the future. For example, a third technique might choose between OPC and DCC for specific blocks of a given function, unlike this work, where each function uses either OPC or DCC exclusively. Other, more exotic techniques could also be developed. However, that is out of the scope of this paper.

7.6.3 General Requirements

While our techniques are implemented specifically on the T-CREST platform using the Patmos ISA and processor, the requirements of single-path code are more general. They can work on any system that adheres to the following:

- It must support single-path code. I.e., at minimum, have conditional moves.
- Instruction timings must be constant except for the cases this work addresses.
- The processor must be fully in-order, i.e., instruction-load timing cannot affect and is not affected by data loading [147].
- Memory accesses must have fixed latency.
- It must be possible to avoid the use of data caches.

For example, Patmos' use of a stack cache is optional for our work. However, its absence would require changes in our implementation to account for register spill/restore instructions. Likewise, Patmos' method cache is optional. Because single-path code always executes the same instruction stream, any deterministic instruction cache can be used. However, there is a performance benefit to using special-purpose caches [148].

7.7 Implementation

We extend the open-source work presented in [17] to produce CET single-path code. Patmos' compiler is based on the LLVM compiler framework [49]. Its frontend, called Clang, produces the LLVM intermediate representation called Bitcode. Bitcode is then compiled by the backend into machine code. The previous work and our extensions all reside in the backend. LLVM also provides Compiler-RT, which is a set of runtime functions. The compiler may assume that these functions are always present and are usually linked into any final program. Compiler-RT includes functions that implement functionality not natively available on the machine. For Patmos, this includes division and floating point operations. In the rest of this section, we highlight the changes we made to the single-path transformation to accommodate the pseudo-root optimization, our scheduler, and achieving CET.

In the rest of the section, we will first describe how we eliminate data cache usage, which is required for achieving CET. We then describe our repetition dominance analysis implementation and how it is used to identify pseudo-roots. Then, we present our simple instruction scheduler, its heuristics, and how it can take advantage of the relaxed restrictions on Patmos' second issue slot. Lastly, we describe how we select between the OPC and DCC compensation techniques and how they are each implemented.

7.7.1 Data Cache Elimination

During instruction selection, the compiler automatically inserts instructions that access non-stack data through the data cache. To ensure the data cache is not used in our implementation, we convert all such loads/store instructions into their counterparts that access the same data directly from the main memory. We can then focus on only these instructions moving forward.

7.7.2 Repetition Dominator Analysis

We have implemented Algorithm 7.2 as a `MachineFunctionPass` in the LLVM backend. It runs our algorithm on each function (in an intermediate representation very close to Patmos machine code). It returns a map from their blocks to the set of blocks that repetition-dominate them.

Canonically, we would need to construct the FCFG of each function before running our algorithm. However, we omit this because LLVM provides `MachineLoopInfo`, which can be used as a substitute. It can be queried for the loop depth of each block, its header, back edges, and exit edges. It allows us to explore the FCFGs without actually having to construct them. We also do not have explicit unilatch nodes. Instead, we return an additional result that contains what would be the unilatch dominators. Uni-exits are also not explicitly needed, as they are required only for equivalence class detection, which explicitly handles exits where relevant. Our implementation also deviates from the algorithm by merging the loop on line 4 with the one on line 7. The results from inner loops are only needed when we reach the header in the outer FCFG. The first time a header is encountered, we run the algorithm on it and use the result immediately. Our deviations from the algorithm do not change the results and only simplify the implementation.

Algorithm 7.2 does not specify how to check for dominators between exits (line 9). We do so by maintaining a map of which nodes are reachable from which other loops—in the FCFG only. We return this map so that it can be used by outer loops in the check. As a result, our function implementing Algorithm 7.2 returns three results, the latter two of which are ignored in the final result.

Lastly, our CFG does not have the ǂ-label. Instead, we provide the algorithm a function that, when given a header, returns whether it should be treated as constant. It does so by looking at the loop iteration bounds; if they are equal, the loop must be constant.

7.7.3 Identifying Pseudo-Roots

Identifying pseudo-roots is done in the `SPMark` pass of the single-path transformation [17]. We update it so that while identifying functions that will be called from a single-path context, it also identifies which calls are coming from a repetition-dominant block and marks the target functions as pseudo-roots.

The `SPReduce` pass handles assigning each instruction a predicate. It also handles converting call instructions not to be predicated and provides the additional predicate argument to the functions. When it sees a call instruction in a repetition-dominant block in a pseudo-root function, it omits the predicate argument, predicates the call instruction, and targets the pseudo-root version of the function (instead of the version that takes a predicate argument.)

7.7.4 Scheduling

We have implemented a simple heuristic, predicate-aware list scheduler that can bundle instructions to take advantage of Patmos' two issue slots. As with all scheduling, it first creates a dependence graph between all the instructions. The graph takes disjoint equivalence classes into account to omit dependencies between disjoint instructions. Each edge in the graph has an associated strength. If the edge is *weak*, the target instruction cannot be scheduled before the source instruction but may be bundled with it. An example would be a branch—which can be in the middle of our blocks at this point—which must be scheduled after all instructions preceding it (control dependency). However, it may also be bundled with any of them. The other instructions in the bundle will be executed before any instructions that control flows to after the branch is taken. If the edge is *strong*, the target instruction can only be scheduled when the source instruction has finished executing. That could be because one instruction needs the output of another, e.g., a loaded value used in succeeding instructions.

The scheduler step-wise chooses the next instruction in the schedule based on the following prioritized heuristics:

1. Instructions with longer delays are always prioritized so that their execution begins as early as possible, so any strongly dependent instructions can become ready for scheduling as soon as possible.

2. Instructions ineligible for the second issue slot are then scheduled so that other instructions might be bundled with them.

3. We then prioritize instruction with a higher number of successors in the dependence graph, which makes more instructions ready for scheduling, hopefully resulting in better scheduling options.

4. Long instructions (i.e., taking up both issue slots) are the penultimate priority, as they cannot be bundled with any other instructions. In a sense, they are already maximally efficient concerning bundling.

5. Lastly, we prioritize earlier instructions in the original schedule. This is a simplification, such that no reordering is done when there is no benefit to doing so. This allows us to analyze the resulting schedules better and reduces random scheduling decisions that might affect our results.

Permissive Dual-Issue

To explore the effects of relaxing the restrictions on the second issue slot, as described in Section 7.5.3, we updated the Patmos cycle-accurate simulator, Pasim, to allow memory accesses, multiplies, and control flow in the second issue slot. If the simulator executes a bundle containing two instructions from the same type, where both are enabled, it will issue an error. Otherwise, it will ignore the disabled one.

We updated the scheduler to match these requirements. Any time it tries to populate the second issue slot in a bundle, it will filter out any non-disjoint instructions that match the type of instruction in the first slot.

7.7.5 Compensation Selection

While the code still uses virtual registers (i.e., before register allocation), we must decide which compensation technique each function should use. The user can supply the compiler with a flag choosing whether all functions should use OPC, all DCC, or “hybrid.” The third option allows the compiler to select between the two compensation techniques for each function.

The selection starts by analyzing the function to find A_{max} and its counterpart, A_{min}: the maximum and minimum number of memory accesses any path may execute, respectively. This analysis is a simple path enumeration: We first count the number of memory-accessing instructions in each basic block. Then, we walk the CFG, summing up access counts and tracking the minimum and maximum accesses. When two paths merge, we update the tracking accordingly. We also scale accesses to the loop bounds such that blocks after loops have the tracking account for the highest and lowest number of iterations the loop will take. We chose this approach for its simple implementation. More complex techniques can be used to reduce compile time, but that is outside the scope of this paper.

If $A_{max} = A_{min}$, all paths access memory the same number of times with no need for compensation. Otherwise, a restricted DCC determines how many instructions would at least be needed to implement it. This is a lower estimate, as more or fewer instructions may be needed depending on register allocation. Then, the main-memory accessing instructions are counted, as the same number of instructions would need to be added by OPC. The heuristic we use to select between the two simply chooses the lowest one. Functions with few accesses, therefore, do not incur the overhead of DCC's compensation loop.

We use pseudo-root and repetition dominance information to optimize A_{min}. For non-pseudo-root functions, A_{min} must be set to zero since a false predicate argument may disable all memory instructions. However, if the function is a pseudo-root, a much better lower bound can be found by looking for the dominant blocks, as they would always be enabled the same number of times, and only when enabled will their memory accessing instructions count. This optimization enables single-path code to account for pseudo-roots to better estimate the variation in accesses. In the best case, the access variation for a function is zero (the number of accesses to memory is always the same), meaning no compensation is needed. For other cases, a lower variation might mean that the CET code will use a different compensation technique to reduce execution time. For the DCC technique specifically, a lower variation means the compensation loop inserted to the end of each function can be given a lower maximum iteration count, reducing the compensation it performs.

7.7.6 Decrementing-Counter Compensation

Any function that has been assigned DCC is immediately transformed. However, because the code is in LLVM's machine code representation, the regular, built-in optimization passes cannot be used. We are, therefore, forced to implement a simplified constant propagation optimization:

We run our compensation on the FCFGs of the function and optimize the decrements as follows: Starting from the header, we forward each edge's label to the edges outgoing from our edge's target block. Say we have blocks a, b, and c, with 3, 6, and 2 loads respectively. $a \to b$ is first labeled with 3 and $b \to c$ with 6. Then, $a \to b$'s 3 is forwarded to $b \to c$, resulting in it being labeled with 9. We then forward the 9 to any of c's outgoing edges. If a block has multiple outgoing edges, they each get forwarded the accesses from the incoming edge. We cannot forward the accesses if a block has multiple incoming edges, as each might differ. Instead, we stop forwarding, decreasing the counter in the predecessors by the amount labeled. If we encounter a back edge, we always stop the forwarding. Thus, each loop iteration decrements the counter by the number of accesses needed. Nothing special is done for the exit edges except to note that the label value will be used when labeling in the outer loop. Remember that inner loops are only represented by their header in the outer loops. Any exit edges are represented by edges from the header to a block in the current loop. When encountering a header, we forward accesses from incoming edges to the outgoing as usual.

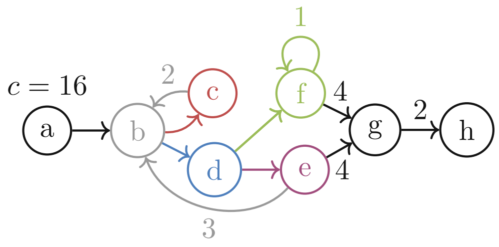


Figure 7.5: Decrementing counter compensation points. We assume the b-loop iterates up to three times and f-loop up to five times.

In Figure 7.5, we have annotated the edges with the result of running DCC on our example program. For simplicity, we assume each block performs only one memory access. We also assume the b-loop may iterate up to three times. The f-loop may iterate five times. That means the program will perform at most 16 memory accesses. That happens when the first loop iterates maximally, exiting through $d \to f$, which also loops maximally before exiting to g. Our technique results in six decrements being needed: The first loop's two back edges $c \to b$ and $e \to b$, the second loop's back edge $f \to f$, the two edges converging on g, and lastly $g \to h$. We have labeled each edge with how much the counter should be decremented. Notice how the decrement for $e \to g$ is based on the one accesses from $a \to b$, which is forwarded to $e \to g$ (since the first loop is only represented by b in the outer-most FCFG) plus the three accesses from b to e. Notice also how both edges to g should decrease by 4. This coincidence can be optimized by forwarding the decrements to the $g \to h$ edge, saving two instructions. However, say f had a different number of accesses, these two decrements would differ and could not be forwarded.

After the optimization, we are ready to insert the compensation code. First, the counter is initialized in the starting block by assigning A_{max} to a virtual register. Then, we add instructions that decrement the counter register at the source block of any labeled edge. Lastly, we need to insert the compensation loop in the final block of the function. To optimize performance, we do not add the loop directly into the function. Instead, we add a dedicated compensation function to Compiler-RT that performs the compensation. This function takes two arguments: the number of memory accesses it needs to compensate for and the maximum number of accesses it may be asked to compensate for $(A_{max} - A_{min})$. The function is then a simple single-path loop that performs several enabled loads equal to the first argument and uses the second argument for loop conversion, such that the function always runs in constant time. In the calling function, we insert a call to the compensation function in the final block. We use this implementation because the compensation function is called from many single-path functions. Its code should often be available in the method cache, making it more efficient to call than to have every function include the code in-line.

7.7.7 Opposite-Predicate Compensation

OPC needs to know which predicates are assigned to each block and instruction. Therefore, the pass implementing it must run after the pass in the single-path transformation that makes predicate assignments (called *Single-Path Info* in [17]). Then, we simply go through all instructions and add our compensation loads with the negated predicates.

7.8 Evaluation

We use a subset of the TACLe benchmark suite to evaluate our work [50]. We include only those programs that successfully compile for all our different compilation configurations and produce the correct result when run on Pasim.[6] We excluded programs for the following reasons: Eight programs had recursive calls, and three had loops without valid bounds. WCET-analyzable code can have neither unbounded recursion nor unbounded loops, so we disregard programs with it. Three programs were compiled incorrectly for traditional code. Fourteen programs either compiled wrong or threw an error during compilation for single-path code. Two programs, `pm` and `filterbank`, compiled and ran correctly but were so long-running that they saturated the Patmos simulator's cycle counter (32-bit). In the end, we have 26 programs to evaluate. Unless otherwise stated, the default single-path configuration uses all repetition dominance optimizations, dual-issue scheduling with PAS without permissive dual-issue, and the hybrid CET technique.

The rest of the section first covers how our repetition dominance optimizations affect the performance and size of single-path code. We then look at the performance of our scheduler and compare it to the scheduler for traditional code. Lastly, we evaluate how enabling CET affects performance. We look at the various compensation techniques, at how the DCC compensation function affects performance, and compare the CET single-path code performance with WCET-analyzed traditional code.

7.8.1 Repetition Dominance Optimizations

Figure 7.6 shows the effect of enabling our repetition-dominance-based pseudo-root optimizations on the performance and size of single-path code. The blue bars show the performance increase ($\frac{disabled - enabled}{enabled} \times 100$) while the red bars show the code size increase in bytes ($\frac{enabled - disabled}{disabled} \times 100$).

Performance

First, note that some programs see little difference in execution time. Some are simple, with only a single function and very little looping. Looking at `statemate` in Table 7.2, we can see that it has only one function (first column) and only a single loop using a counter to manage iteration (last column). Our optimizations make little difference, as

[6] All compilations use Clang's `O2` optimization level, the highest our implementation supports.

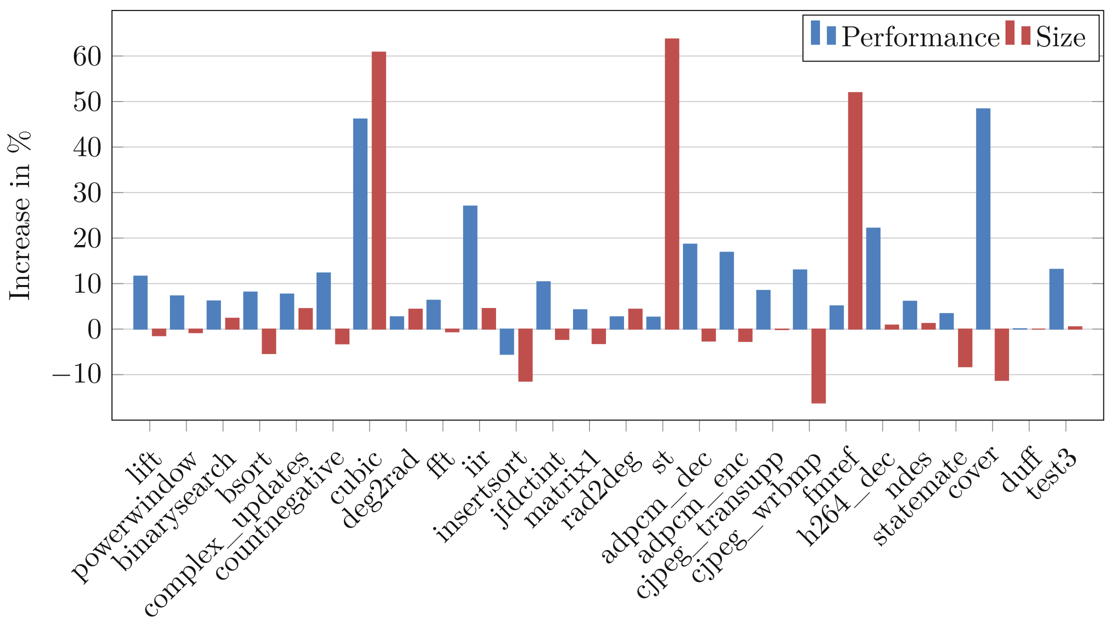


Figure 7.6: Performance and size increase of enabling pseudo-root optimizations for single-path code.

there are no non-root functions to designate as pseudo-roots, and the looping, at best, avoids one counter.

We see significant performance increases for some of the more complex programs. In the best case, `cubic` sees a 46 % performance increase. This number is all the more impressive when we look at Table 7.2. First, notice that the number of functions increases from 33 to 47. Notice also, in the second column, that the number of pseudo-roots found was 17 (including the root). Thus, 14 pseudo-roots are also used in a non-pseudo-root context, which means two copies of each original function must be used. The rest of the functions are either only used in a pseudo-root context (3) or in a non-pseudo-root context (16). So, from where does all that performance come? The source code shows that the main function is four constant loops nested within each other. The function `cubic_solveCubic` is called four times before the loop and once in each iteration of the inner-most nested loop. Cumulatively, the main function has 879 calls to this function, all from repetition-dominant blocks. Therefore, recognizing `cubic_solveCubic` exclusively as a pseudo-root likely produces most of this substantial increase in performance.

For the `insertsort` program, we see a 5.5 % performance reduction even though it also only has one function. However, looking at the third and fourth columns of Table 7.2, we can see that enabling our optimizations results in using OPC instead of DCC. That is because the more accurate memory access counts mean fewer instructions are needed to compensate. The heuristic used to choose between OPC and DCC selects the one with

Table 7.2: Compiler statistics for each program using single-path code. For each entry, repetition dominance optimizations are disabled for the number on the left and enabled for the one on the right. The metrics given are the total number of functions, the number of pseudo-root (PR) functions, the number of functions using OPC, the number of functions using DCC, and the total number of loops using a counter.

	Functions		Compensation		Loop
	Total	PR	OPC	DCC	Counters
lift	1 | 1	1 | 1	0 | 0	1 | 1	6 | 1
powerwindow	5 | 5	1 | 3	2 | 0	3 | 3	3 | 0
binarysearch	1 | 1	1 | 1	1 | 1	0 | 0	1 | 1
bsort	1 | 1	1 | 1	1 | 1	0 | 0	2 | 1
complex_up..	21 | 23	1 | 10	11 | 6	7 | 7	1 | 0
countnegative	1 | 1	1 | 1	0 | 0	0 | 0	2 | 1
cubic	33 | 47	1 | 17	17 | 16	12 | 16	6 | 5
deg2rad	27 | 30	1 | 12	13 | 9	11 | 11	1 | 0
fft	1 | 1	1 | 1	0 | 0	1 | 1	6 | 4
iir	21 | 23	1 | 10	11 | 6	7 | 7	1 | 0
insertsort	1 | 1	1 | 1	0 | 1	1 | 0	2 | 1
jfdctint	1 | 1	1 | 1	0 | 0	0 | 0	2 | 0
matrix1	1 | 1	1 | 1	0 | 0	0 | 0	3 | 2
rad2deg	27 | 30	1 | 12	13 | 9	11 | 11	1 | 0
st	34 | 48	1 | 18	18 | 16	12 | 16	4 | 1
adpcm_dec	3 | 3	1 | 3	1 | 1	1 | 0	6 | 4
adpcm_enc	3 | 3	1 | 3	1 | 2	2 | 0	7 | 5
cjpeg_transupp	1 | 1	1 | 1	0 | 0	1 | 1	56 | 51
cjpeg_wrbmp	3 | 3	1 | 3	1 | 1	1 | 0	4 | 2
fmref	72 | 103	1 | 36	25 | 29	43 | 50	11 | 6
h264_dec	1 | 1	1 | 1	0 | 0	0 | 0	7 | 6
ndes	1 | 1	1 | 1	0 | 0	1 | 1	13 | 10
statemate	1 | 1	1 | 1	0 | 1	1 | 0	1 | 0
cover	1 | 1	1 | 1	0 | 0	0 | 0	3 | 0
duff	1 | 1	1 | 1	0 | 0	0 | 0	0 | 0
test3	101 | 101	1 | 101	0 | 0	100 | 0	121 | 0

the fewest instructions added to the function. That is clearly sub-optimal in this case, as DCC is the best choice, even if it uses more instructions. Configuring the compiler to choose DCC exclusively verifies this, as we see a performance increase of 4.7 %

instead. To avoid performance reductions like these, a better heuristic is needed that more accurately predicts which compensation technique is superior for a given function.

We can look at `test3`'s numbers to clearly see the beneficial effects of more accurate memory access counts. Without our optimizations, 100 functions used DCC to compensate for memory access variability. However, since all functions can be recognized as pseudo-roots, our optimization allows the compiler to recognize the lack of variability in memory accesses, meaning no functions need compensation.

Code Size

Our optimizations can significantly affect code size. It may increase when a function is used in both pseudo-root and non-pseudo-root contexts. That results in two copies of the original function. However, when a function is exclusively a pseudo-root, it will require fewer instructions to manage its predicates, and some of its loops may avoid using an iteration counter. Memory access compensation should also use fewer instructions with our optimizations enabled. First, if functions are recognized as not needing compensation, no instructions are necessary to ensure fixed memory access counts. Second, repetition-dominant blocks do not need their memory accesses compensated for, meaning both compensation techniques will use fewer instructions to compensate pseudo-roots. Lastly, our optimizations may make the compiler choose a different compensation technique. That should reduce code size, as the heuristic specifically uses the number of instructions added to choose between the techniques.[7]

In Figure 7.6, we can see that the `cubic`, `st`, and `fmref` programs see the biggest increase in code size (61 %, 64 %, and 52 %, respectively). The increase in code size can be roughly attributed to the fact that around half of their functions are used in both pseudo-root and non-pseudo-root contexts. While `cubic` sees good performance gains, the two other programs do not. `st` specifically sees almost no benefit from the optimization, which means the increase in code size negates almost all the performance improvement the optimization might provide. Some analysis should be performed to identify functions that would not benefit from using a pseudo-root version; e.g., the pseudo-root version might not be called enough to justify the increased pressure on the method cache. This should both reduce code size and potentially produce further performance increases.

Several programs see significant code size decreases, mostly when functions are exclusively either pseudo-roots or not. For example, `cjpeg_wrbmp` has three pseudo-root functions, where only one of them needs compensation when our optimizations are enabled. This results in the largest decrease in code size of 19 %.

[7]Switching compensation techniques may also increase code size, as the heuristic can only estimate how many instructions DCC will add.

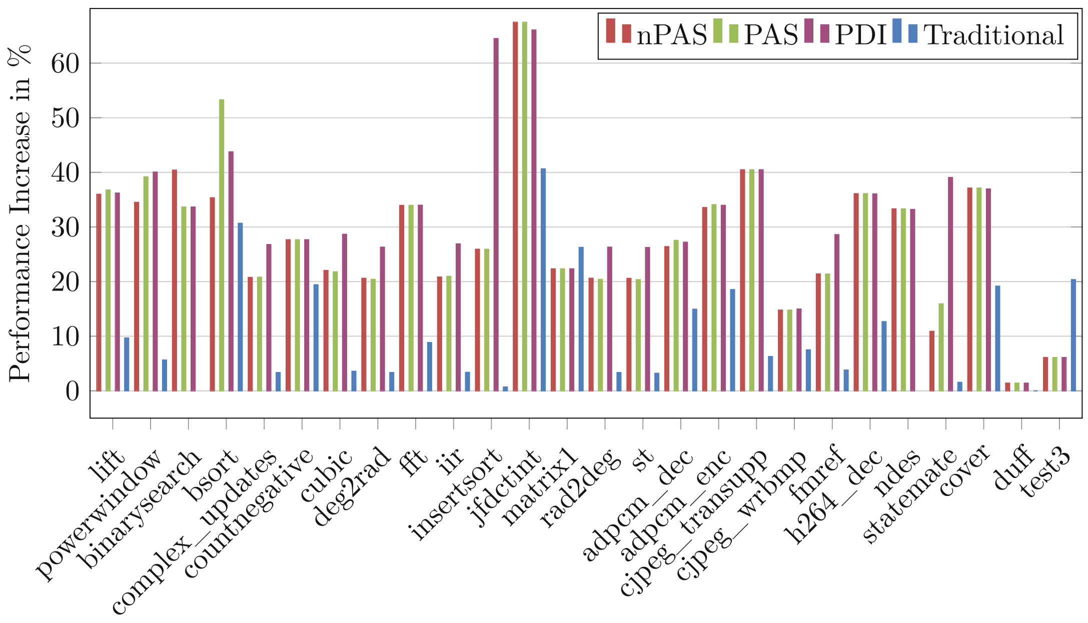


Figure 7.7: Performance increase of enabling dual-issue scheduling (minus main-memory stalls).

7.8.2 Dual-Issue Scheduling

In Figure 7.7, we investigate the effectiveness of our scheduler in utilizing the second issue slot of Patmos. To isolate the scheduling performance, we subtract from the execution time of each program the number of cycles that were spent stalled, waiting for main memory. Otherwise, the execution time would be dominated by memory-related stalls, and the effect of scheduling would be difficult to see. In the red, green, and purple bars, we compare the performance increase of enabling dual-issue scheduling without PAS (nPAS), with PAS (PAS), and with permissive dual-issue (PDI), respectively. The baseline is the same scheduler using only one issue slot without PAS or permissive dual-issue.

Predicate-Aware Scheduling

We generally do not see significant performance improvements when enabling PAS. The outlier is the `bsort`, where we see a 13 % performance increase compared to when the scheduler is not using PAS. This increase is only due to the additional information allowing the scheduler to find one additional bundle and eliminating one no-op instruction in the 21-instruction main loop.

For the `binarysearch` program, we see a 5 % decrease in performance. Looking at the assembly code shows that enabling PAS results in one less bundle in the 23-instruction main loop. This is an artifact of our scheduler's simplicity. The difference in instruction dependencies causes the scheduler to make different decisions early on that produce adverse results later. As far as we can tell, the nPAS schedule cannot be improved for this program, even with PAS. A more sophisticated scheduler should avoid this decrease in performance.

The benefits of PAS in scheduling are, at best, sporadic. However, since this information always results in fewer scheduling dependencies between instructions, more sophisticated schedulers should be able to extract more benefits. Architectures with more issue slots and functional units could also benefit from the increased bundling opportunities. So, while the benefits we have seen are limited, we still believe any practical single-path scheduler should use PAS.

Permissive Dual-Issue

Enabling permissive dual-issue gives slightly better results than using PAS exclusively. We see seven programs that get ∼5 % performance benefit. Additionally, we see `statemate` and `insertsort` get a 20 % and 31 % increase, respectively, on top of any increase from enabling dual-issue scheduling and PAS. Looking at the assembly of `insertsort`, the performance increase comes from bundling compensatory load instructions of OPC with the loads or stores they compensate for. This increases the number of bundles in the 22-instruction main loop from six to ten, only one bundle less than the theoretical maximum.

The results of enabling permissive dual-issue reinforce the need for PAS in single-path code scheduling. Similarly to the PAS on its own, permissive dual-issue is likely more beneficial with more sophisticated schedulers and architectures with more resources and issue slots. However, it is still an open question whether the hardware resources needed to support permissive dual-issue are worth it. We do not expect permissive dual-issue support to prohibitively increase hardware complexity, but we have made no effort to study it as part of this work.

Comparison with Traditional Scheduling

In the blue bars of Figure 7.7, we have added the performance increase of enabling dual-issue scheduling for traditional code. This is calculated using the WCET bounds provided by Platin [149], minus the maximum number of cycles Platin estimates will be stalls waiting for main memory. We have excluded the `binarysearch` program because the analyzer produced an incorrect WCET bound and the `ndes` program because Platin failed to provide a bound for the single-issue code.

Compared to our single-path code, we can see that the traditional scheduler often makes significantly less use of the second issue slot. The scheduler used in this case is based on LLVM's built-in scheduler. Because the code has not been if-converted, there is less ILP to exploit. For example, the `binarysearch` program essentially has no ILP without if-conversion, meaning the traditional scheduler can never do better. In contrast, single-path code easily takes advantage of the ILP introduced by if-conversion, resulting in much more substantial use of the second issue slot for most programs evaluated. `test3`

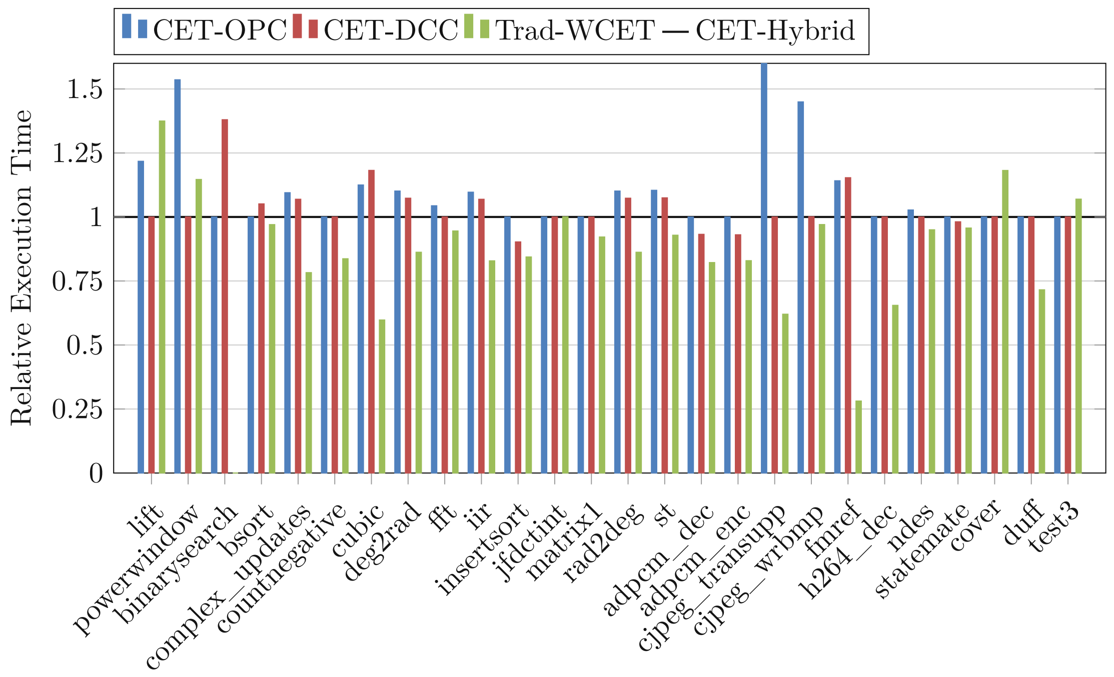


Figure 7.8: Execution time comparison for each configuration. `CET-Hybrid`'s execution cycles is the baseline at 1.

is an outlier, seeing much more benefit for the traditional code than single-path. One reason for this could be the program's simplicity, which is mainly comprised of loops. This intrinsically results in big basic blocks for both traditional and single-path code. Without the inherent advantage of single-path code's higher ILP, the simplicity of the single-path scheduler could be exhibited here, compared to the traditional code's more mature scheduler.

7.8.3 Constant Execution Time

To get a sense of how our two compensation techniques for achieving CET perform, we compare the execution time of the benchmark programs using three different compilation configurations: (1) Using OPC for all functions (`CET-OPC`), (2) using DCC for all functions (`CET-DCC`), and (3) using the hybrid approach described in Section 7.7 (`CET-Hybrid`). Figure 7.8 shows the relative execution times in cycles, where `CET-Hybrid` is the baseline at 1. A bar higher than 1 signifies that the configuration requires more cycles to execute. A bar lower than 1 means it requires fewer cycles to execute. E.g., for `binarysearch`, `CET-OPC` requires the same number of cycles as `CET-Hybrid` (both 566), while `CET-DCC` requires more cycles (781). Note that the `CET-OPC` configuration of `cjeg_transupp` is out of bounds at 3 times the execution time of `CET-Hybrid`.

The first thing to note is that `CET-OPC` and `CET-Hybrid` have the same execution time

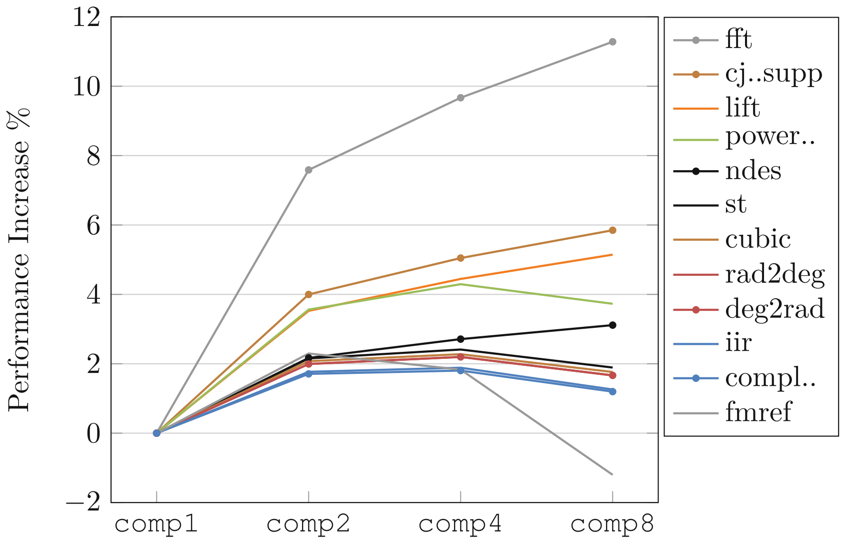


Figure 7.9: Performance increase gained by using different compensation functions in DCC. The increases are relative to using the `comp1` compensation function.

for 12 programs. E.g., `binarysearch` and `bsort`. Looking at Table 7.2's third and fourth columns, we can see the compilation data of `CET-Hybrid` and the distribution of functions that used OPC or DCC in each program. When the sum of the two columns does not sum to the total number of functions in the first column, the missing functions need no compensation. We confirm that the 12 programs had all their functions needing no compensation or using OPC exclusively. For most programs, `CET-Hybrid` was the most performant configuration (or at least equal to the others.) Therefore, it seems `CET-Hybrid`'s simple heuristic choosing between OPC and DCC is mostly acceptable. However, there is room for improvement since we can see that, e.g., `insertsort`'s `CET-Hybrid` execution is higher than `CET-DCC`. The heuristic assigns the one function to use OPC, which is clearly sub-optimal. Using DCC, in this case, results in a 11 % performance increase.

If we look at the programs that used a mix of OPC and DCC, we see no clear favoring of one compensation technique over the other. This confirms that it is beneficial to have a compensation technique with minimal overhead for cases where little compensation is needed and one for higher access count variation.

Our results show that a minimal-overhead compensation technique is critical to performance. However, `CET-OPC` does add significant overhead by introducing new load instructions. It is possible to avoid this overhead if the hardware can assist. While Patmos does not support this, we can imagine having a mode-switch that dictates whether disabled memory accesses query main memory. That would allow functions assigned `CET-OPC` to enable that mode instead of adding extra loads. However, `CET-DCC` functions would disable it to avoid incurring the main memory latency too many times. Our results show that a core that always triggers memory access latency on disabled instructions (with no facility for the program to control it) would perform worse, as `CET-DCC` would not be available for high-variability functions or functions where many paths execute few memory accesses each. Take `cjpeg_wrbmp`, where `CET-Hybrid` provides a 45 % increase over `CET-OPC`. Looking at the runtime statistics from the simulator, we see that `CET-OPC` performed 17 653 requests to main memory. Since each request is performed by one of two memory instructions (either the original or the compensation), at most the same number of cycles could be saved by hardware assistance.[8] Reducing the WCET bound by the number of added compensation instructions still leaves an increase of 39 % for `CET-Hybrid` without hardware assistance.

DCC Compensation Function

Since DCC uses a compensation function at the end of all regular functions, it is interesting to look at how its implementation affects performance. To investigate that, we implemented four different compensation functions in Patmos assembly. They are all primarily comprised of a single-path loop, where either 1, 2, 4, or 8 compensatory loads are performed in each iteration. We will refer to these functions as `comp1`, `comp2`, `comp4`, and `comp8`, respectively.

As we increase the number of loads per iteration, performance should increase as it amortizes the overhead of the looping. E.g., `comp1` requires 6 instructions in 4 bundles per iteration while `comp4` requires 9 instructions in 6 bundles. However, extra code outside the loops must be added to handle the remainder to ensure all functions can handle non-power-of-two compensation amounts. `comp2` only requires two additional instructions for this, `comp4` requires 9, while `comp8` has a full copy of `comp2` following its primary loop, as that requires fewer instructions than inlining it all (10 versus 18 instructions). Our four compensation functions' instruction/bundle counts are 6/5, 10/7, 16/11, and 27/18, respectively. Note that the execution times in Figure 7.8 all use `comp4`.

In Figure 7.9, we see the speedup gained by using the different compensation functions. We include the programs whose `CET-Hybrid` used DCC for at least one function. We can see that most functions benefit from "bigger" compensation functions. The most significant improvement happens between `comp1` and `comp2`, where we see gains from 1.5 % to 7.5 %. Another less than 1 % is then generally gained by using `comp4`, while a few gain considerably more, e.g., `fft`. Using `comp8` brings marginal additional increase for most, with outliers either gaining significantly more (`fft` for a total gain of 11.3 %) or seeing reduced performance (`fmref`). Multiple reasons for `comp8`'s lackluster performance may exist. First, since many regular functions call the compensation function, the latter is often needed in the method cache. Therefore, the larger they are, the bigger the chance they force more cache evictions for the other functions. Additionally, the performance of our different compensation functions varies based on

[8]Some of these requests may come from the method or stack cache, and so this number is an upper bound on the number of instructions that might benefit from hardware support for compensation.

the exact maximum compensation needed for each single-path function. E.g., `comp8` has the worst performance at each exact power of 8 because it is needlessly iterating in its secondary loop.

7.8.4 Comparison to WCET Bounds

Since CET can be used in place of WCET bounds produced by an analyzer, it is interesting to see how our work performs compared to the Platin analyzer. We include the WCET bounds produced by Platin in Figure 7.8 with the name `Trad-WCET`. As mentioned, we exclude the erroneous bound for `binarysearch`. Platin produced a bound of 260. However, actually running the program requires 348 cycles. Clearly, Platin produced an invalid bound.

Looking at the rest of the WCET bounds, the general trend is that they are lower than our best configuration, some more than others. However, `lift`, `powerwindow`, `jfdctint`, `cover`, and `test3` have lower execution times using single-path code compared to the bound. This shows that single-path code can be superior to static analysis, especially where single-path code's weaknesses are less prevalent. `test3` is such a case, as it has minimal branching and fixed-iteration loops. `lift` gets the largest gain from using single-path code at 37 %, even though it is not as obviously simple as `test3`. For example, it has some functions with deep if-else nesting, which should increase the single-path overhead. Its heavy use of constant loops might be what makes single-path code incur little overhead, and standard LLVM optimizations might be minimizing the amount of branching.

The Platin analysis of `lift` shows that it expects up to 3 556 098 cycles to be spent on servicing data cache misses in the traditional code.[9] In contrast, single-path code spends 3 517 836 cycles stalling on all main memory accesses, not only for servicing random accesses but also for fetching instructions and stack data. That means that Platin's estimate on how often the cache is accessed (and its conservative assumption that they all miss) significantly contributes to the poor WCET bound.

The method cache analysis exacerbates this. Platin estimates 1 893 780 stall cycles being spent on servicing method cache misses, translating to 90 180 additional instructions fetched from main memory. In contrast, single-path code only experiences seven method cache misses. Each method cache miss results in a predefined number of bytes (a section of code) being moved to the method cache. In the case of single-path `lift`, the largest block is 252 instructions. I.e., at worst, those missed would result in $252 \times 7 = 1764$ instructions being fetched from main memory, translating to 37 044 stall cycles. In reality, single-path `lift` only spends 3822 cycles stalled by the method cache. It seems Platin is not very good at determining what combination of code blocks will likely be in the method cache at a given time. It both highly overestimates the number of misses and is overly pessimistic about the size of the blocks fetched for each miss.

[9]A cache-miss incurs a 21-cycle latency similarly to accessing main memory directly.

```
if COND then                    if COND then
  func(1)                         a = 1
else                            else
  func(2)                         a = 2
end if                          end if
                                func(a)
```

Figure 7.10: Functions called from all branches (left) can be extracted and only called from one position (right). This figure was originally featured in [150] as Figure 4.8.

In addition to the six programs that are outright faster when using single-path code (including `binarysearch`) we can classify a further eight as having comparable performance (less than 10 % slower) and another six as having competitive performance (less than 20 % slower). We are optimistic about these results, showing that most programs (20 out of 26) have competitive, comparable, or outright better performance than using a WCET analyzer on traditional code. We can imagine an application where a performance penalty of, e.g., 20% would be acceptable in exchange for the simplicity of CETs.

7.9 Further Optimizations

In this work, we have presented optimizations based on repetition dominance and PAS. The results show impressive improvements in some cases, which leads us to believe additional significant improvements can be found. Because single-path code's primary challenge is its execution-time overhead, it is essential to explore all avenues of optimization.

Platzer analyzed the limitations of single-path code (i.e., which types of code produce large overheads) and identified accessible optimization opportunities [150]. One such opportunity is identifying functions called from both cases in a branch, as seen in Figure 7.10. In the current implementations of single-path code, the function would be called twice, once enabled and once disabled. That is unnecessary. Regardless of the path taken, the function must be called only once. The optimization would recognize such cases and instead perform the call after the paths converge. Instead of the call, the two paths would assign their respective call arguments to variables, which are then passed to the single call point. Platzer also mentioned that this type of optimization could be made for other scenarios, like reading from an array based on an index.

We see Platzer's proposals as an instance of a larger set of optimization opportunities around branches. Recognize that only one branch path will ever be executed at a time. Using similar instruction sequences is also quite ordinary for two paths. Platzer's example is one such, where both paths need to call a function. Another possibility is that both paths need to load some memory and will include a sequence calculating the address and issuing the load. In Figure 7.11, we see a sequence of instructions in single-path code.

```
cmp             p1 = r5 , r6           cmp             p1 = r5 , r6
add   ( p1)     r1 = r2 , r3           add   ( p1)     r1 = r2 , r3
mul   ( p1)     r3 = r1 , r4   <-      sub   (!p1)     r1 = r2 , r3
add   ( p1)     r1 = r2 , r3           mul             r3 = r1 , r4   <-
shr   ( p1)     r3 = r1 , 2    <-      add   ( p1)     r1 = r2 , r3
sub   (!p1)     r1 = r2 , r3           sub   (!p1)     r1 = r2 , r3
mul   (!p1)     r3 = r1 , r4   <-      shr             r3 = r1 , 2    <-
sub   (!p1)     r1 = r2 , r3
shr   (!p1)     r3 = r1 , 2    <-
```

Figure 7.11: Instructions shared between different branch paths (left) can be coalesced into just one copy used by both (right).

The branch condition is used to predicate the instructions for both paths. We see an optimization opportunity in identifying shared sequences of instructions between branch paths and coalescing them such that only one copy is needed. Here, we have highlighted the shared instructions using arrows. The coalesced sequences would be predicated on the path preceding the branch instead of its condition to ensure its instructions are executed regardless of which branch is needed. The figure shows how the instructions are reordered to fit regardless of the condition, with the shared instructions not being predicated (because the condition is not either.)

Additional optimizations on constant loops can also be found. Take those blocks in the loop that traditionally dominate all exits. These blocks are always enabled, even in the final iteration of the loop. That means they never need to be predicated—as they currently are. Such an optimization could reduce the pressure on the predicate registers. Likewise, take the blocks that are traditionally dominated by all exits. These blocks are always disabled on the final iteration. As such, an optimization could be to skip them in the final iteration only, reducing execution time.

Using the work presented here could also help optimize variable loops. Given a lower bound, we know the loop must iterate at least that many times. An optimization could be to split the loop iterations into two consecutive loops. The first copy would be a constant loop iterating the same number of times as the lower bound of the original loop. The second copy would then iterate the remaining (variable) number of times. That would open up the constant loop to being optimized using repetition dominance. Loop overhead can also be reduced by looking for loops with disjoint equivalence classes. Two such loops could be merged into one, such that the counter-management overhead is reduced, additional ILP is unlocked, and the optimizations above have more instructions to work with. Our method of finding disjoint classes directly applies to this use case, unlike other works that cannot compare predicates across loop bounds [105].

Lastly, more effort can be put into enabling single-path code to use annotation languages. These have been specifically designed to provide information that can be used to find all the feasible paths in a program [63]. That could be useful in providing optimizations to single-path code as well. For example, nested loops whose iteration depends on an outer-loop counter have been explored for tighter WCET bounds [66]. That could also be useful in single-path code, where these inner loops could be treated as constant loops as long as their outer loops are also constant (and the inner loops are not on a branched path.)

7.10 Conclusion

For any practical application of single-path code, two things must hold: 1) Its performance must be comparable to that of statically analyzed traditional code and 2) it must be guaranteed to have CET, such that static WCET analysis is unnecessary. In this paper, we have addressed these points through three main topics. First, we presented the repetition dominance relation, a variation of the traditional dominance relation on CFGs that also accounts for the number of node visits. Using that relation, we presented the pseudo-root optimization that reduces the overhead of single-path code by reducing the amount of unnecessary code executed. Our results show that optimizing pseudo-roots can improve performance significantly, with observations of up to 48 %. However, the optimization also significantly affects the code size, with an observed increase of 64 % but also a case of a 19 % reduction.

We then discussed how single-path code has unique characteristics that should be accounted for when scheduling instructions for a dual-issue pipeline. We also presented a possible extension to the Patmos ISA, called permissive dual-issue, allowing more instructions to be scheduled in the second issue slot. The performance improvements we saw using these techniques were sporadic but with a best case of 31 %. However, we expect better schedulers and more complex architectures to see additional benefits from using PAS and permissive dual-issue scheduling.

Lastly, we addressed how single-path code does not guarantee CET in the presence of multi-cycle memory access latencies and predicated memory accessing instructions. We presented two memory access compensation techniques, OPC and DCC, that ensure memory is accessed the same number of times regardless of runtime conditions. Our results showed that a combination of these two techniques should be used for optimal performance.

We compared single-path code's performance to traditional code's analyzed WCET bounds. In 20 out of 26 cases, single-path code had competitive performance to traditional code (within 20 %). In 8 of those, the performance was comparable (within 10 %). Most notably, in six cases, single-path code was outright superior, being in the best case 37 % faster than traditional code. Finally, we discussed further optimization opportunities that should be explored to maximize single-path code's performance and reduce the prevalence of pathological cases of poor performance.

Source Access

Patmos and its platform, T-CREST [15], are available as open-source and include the contributions of this paper. The Patmos homepage can be found at https://patmos.compute.dtu.dk/ and provides a link to the Patmos Reference Handbook [51], which includes build instructions.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-llvm-project. [10]

[10]Commit hash: 6460195495d7ca02d3001d84b1a9677328be317f.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Jan Gustafsson and Andreas 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)*, pages 382–392. IEEE, 2007.

[2] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 249–258. IEEE, 2005.

[3] Peter Puschner. The single-path approach towards WCET-analysable software. In *IEEE International Conference on Industrial Technology, 2003*, volume 2, pages 699–704. IEEE, 2003.

[4] Stephen Alstrup, Dov Harel, Peter W Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[5] Adam L Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E Tarjan, and Jeffery R Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

[6] Dov Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 185–194, 1985.

[7] Steven Miller, Elise Anderson, Lucas Wagner, Michael Whalen, and Matts Heimdahl. Formal verification of flight critical software. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 6431, 2005.

[8] Filippo De Florio. *Airworthiness: An introduction to aircraft certification and operations*. Butterworth-Heinemann, 3rd edition, 2016.

[9] Norman K Denzin and Yvonna S Lincoln. *The SAGE handbook of qualitative research*. SAGE, 5th edition, 2018.

[10] Aleksandar Rodić, Gyula Mester, and Ivan Stojković. Qualitative evaluation of flight controller performances for autonomous quadrotors. In *Intelligent Systems: Models*

and Applications: Revised and Selected Papers from the 9th IEEE International Symposium on Intelligent Systems and Informatics SISY 2011, pages 115–134. Springer, 2013.

[11] Tracy L Lamb, Keith J Ruskin, Stephen Rice, Leili Khorassani, Scott R Winter, and Dothang Truong. A qualitative analysis of social and emotional perspectives of airline passengers during the covid-19 pandemic. *Journal of Air Transport Management*, 94:102079, 2021.

[12] Amnon H Eden. Three paradigms of computer science. *Minds and machines*, 17:135–167, 2007.

[13] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[14] Martin Schoeberl et.al. Patmos: a time-predictable dual-issue microprocessor. 2011.

[15] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[16] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[17] Daniel Prokesch, Stefan Hepp, and Peter Puschner. A generator for time-predictable code. In *Proceedings of the 17th IEEE Symposium on Real-time Distributed Computing (ISORC 2015)*, Auckland, New Zealand, April 2015. IEEE.

[18] Michael Platzer and Peter Puschner. A real-time application with fully predictable task timing. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 43–46. IEEE, 2020.

[19] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Compiling for time-predictability with dual-issue single-path code. *Journal of Systems Architecture*, 118:102230, 2021.

[20] Michael Platzer and Peter Puschner. A processor extension for time-predictable code execution. In *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 34–42. IEEE, 2021.

[21] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[22] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[23] Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pages 165–172. IEEE, 2002.

[24] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 13–22. IEEE, 2008.

[25] Jaekwon Lee, Seung Yeob Shin, Shiva Nejati, Lionel C Briand, and Yago Isasi Parache. Schedulability analysis of real-time systems with uncertain worst-case execution times. *arXiv preprint arXiv:2007.10490*, 2020.

[26] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback–feedforward scheduling of control tasks. *Real-Time Systems*, 23(1):25–53, 2002.

[27] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *The 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 57–68, Austin, Texas, USA, 20-24, June 2009. ACM.

[28] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

[29] Rhan Ha and Jane WS Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *14th international conference on distributed computing systems*, pages 162–171. IEEE, 1994.

[30] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[31] K. Hong and J. Leung. On-line scheduling of real-time tasks. In *Proceedings. Real-Time Systems Symposium*, pages 244–250, Los Alamitos, CA, USA, dec 1988. IEEE Computer Society.

[32] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760, 2004.

[33] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, symantec corp., security response*, 5(6):29, 2011.

[34] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[35] Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle.(2012). *Google Scholar*, 2012.

[36] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[37] Jeroen V Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–20, 2012.

[38] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, pages 159–176. Springer, 2012.

[39] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[40] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[41] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.

[42] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[43] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[44] Martin Schoeberl. Design of a time-predictable multicore processor: The T-CREST project. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 909–912, March 2018.

[45] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE.

[46] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[47] Martin Schoeberl, David VH Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 53–62, Madrid, Spain, July 2014.

[48] Emad Jacob Maroun, Henrik Enggaard Hansen, Andreas Toftegaard Kristensen, and Martin Schoeberl. Time-predictable synchronization support with a shared scratchpad memory. *Microprocessors and Microsystems*, 64:34 – 42, 2019.

[49] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

[50] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[51] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

[52] Salvador Lucas. The origins of the halting problem. *Journal of Logical and Algebraic Methods in Programming*, 121:100687, 2021.

[53] Alfred V Aho, Jeffrey D Ullman, et al. *Principles of compiler design.* Addision-Wesley Pub. Co., 1977.

[54] Edward S Lowry and Cleburne W Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[55] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[56] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[57] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. *ACM SIGPLAN Notices*, 33(5):97–105, 1998.

[58] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[59] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 253–265, 1973.

[60] Paul W Purdom Jr and Edward F Moore. Immediate predominators in a directed graph [h]. *Communications of the ACM*, 15(8):777–778, 1972.

[61] Robert Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[62] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[63] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software & Systems Modeling*, 10(3):411–437, 2011.

[64] Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for wcet analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[65] Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. orange: A tool for static loop bound analysis. In *Proceedings of the Workshop on Resource Analysis*, volume 42, 2008.

[66] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[67] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[68] Dimitar Kazakov and Iain Bate. Towards new methods for developing real-time systems: Automatically deriving loop bounds using machine learning. In *2006 IEEE Conference on Emerging Technologies and Factory Automation*, pages 421–428. IEEE, 2006.

[69] Emad J Maroun, Martin Schoeberl, and Peter Puschner. Compiler-directed constant execution time on flat memory systems. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023.

[70] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 382–391. Springer Berlin / Heidelberg, 2012.

[71] Jun Yan and Wei Zhang. A time-predictable VLIW processor and its compiler support. *Real-Time Systems*, 38(1):67–84, 2008.

[72] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Instruction-level parallel processors*, pages 161–170. IEEE Computer Society Press, 1995.

[73] Morteza Mohajjel Kafshdooz, Mohammadkazem Taram, Sepehr Assadi, and Alireza Ejlali. A compile-time optimization method for wcet reduction in real-time embedded systems through block formation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–25, 2016.

[74] Xuesong Su, Hui Wu, and Jingling Xue. Wcet-aware hyper-block construction for clustered vliw processors. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 110–122, 2019.

[75] Bekim Cilku, Wolfgang Puffitsch, Daniel Prokesch, Martin Schoeberl, and Peter Puschner. Improving performance of single-path code through a time-predictable memory hierarchy. In *Proceedings of the 20th IEEE International Symposium on Real-Time Computing (ISORC 2017)*, pages 76–83, Toronto, Canada, May 2017. IEEE.

[76] Bekim Cilku, Roland Kammerer, and Peter Puschner. Aligning single path loops to reduce the number of capacity cache misses. *ACM SIGBED Review*, 12(1):13–18, 2015.

[77] Clemens B Geyer, Benedikt Huber, Daniel Prokesch, and Peter 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)*, pages 1–8. IEEE, 2013.

[78] Albrecht Kadlec, Raimund Kirner, and Peter Puschner. Code transformations to prevent timing anomalies. *International Journal of Computer Systems Science & Engineering*, 26(6):463–479, 2011.

[79] Ayman K Gendy and Michael J Pont. Towards a generic "single-path programming" solution with reduced power consumption. In *ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC-CIE2007)*, volume 4, pages 65–71, 2007.

[80] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[81] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

[82] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[83] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Systems*, 2013.

[84] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[85] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Towards dual-issue single-path code. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 176–183, 2020.

[86] Michael Platzer and Peter Puschner. An instruction filter for time-predictable code execution on standard processors. In *International Conference on Computer Safety, Reliability, and Security*, pages 111–122. Springer, 2020.

[87] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE transactions on computers*, 30(07):478–490, 1981.

[88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on computers*, 37(8):967–979, 1988.

[89] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: An effective technique for VLIW and superscalar compilation. In *Instruction-Level Parallelism*, pages 229–248. Springer, 1993.

[90] Roberto Castañeda Lozano and Christian Schulte. Survey on combinatorial register allocation and instruction scheduling. *ACM Computing Surveys (CSUR)*, 52(3):1–50, 2019.

[91] Srinivas Mantripragada, Suneel Jain, and Jim Dehnert. A new framework for integrated global local scheduling. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pages 167–174. IEEE, 1998.

[92] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, Lyngby, Denmark, May 2012. IEEE.

[93] Edgar Lakis and Martin Schoeberl. An SDRAM controller for real-time systems. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[94] Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pages 55–64, New York, NY, USA, 2013. ACM.

[95] Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2012)*, October 2012.

[96] Daniel Prokesch, Benedikt Huber, and Peter Puschner. Towards automated generation of time-predictable code. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2014.

[97] Gerolf F Hoflehner. Strategies for predicate-aware register allocation. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pages 185–204. Springer, 2010.

[98] Jens Knoop and Oliver Rüthing. Constant propagation on predicated code. *J. Univers. Comput. Sci.*, 9(8):829–872, 2003.

[99] Richard Johnson and Michael Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 100–113. IEEE, 1996.

[100] Jens Knoop, Jean-Francois Collard, and Roy Dz-ching Ju. Partial redundancy elimination on predicated code. In *Static Analysis: 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29-July 1, 2000. Proceedings 7*, pages 260–279. Springer, 2000.

[101] Fabrice Rastello and Florent Bouchez-Tichadou, editors. *SSA-based Compiler Design*. Springer, 2022.

[102] Arthur Stoutchinin and Francois De Ferriere. Efficient static single assignment form for predication. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 172–181. IEEE, 2001.

[103] Lori Carter, Beth Simon, Brad Calder, and J Ferrante. Predicated static single assignment. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*, pages 245–255. IEEE, 1999.

[104] John W Sias, Wen-Mei W Hwu, and David I August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 112–123, 2000.

[105] Mikhail Smelyanskiy, Scott A Mahlke, Edward S Davidson, and H-HS Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 169–178. IEEE, 2003.

[106] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, 1983.

[107] David I August, Wen-mei W Hwu, and Scott A Mahlke. A framework for balancing control flow and predication. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 92–103. IEEE, 1997.

[108] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. *ACM SIGMICRO Newsletter*, 23(1-2):45–54, 1992.

[109] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. Hyperblock formation: a power/energy perspective for high performance vliw architectures. In *2005 IEEE International Symposium on Circuits and Systems*, pages 4090–4093. IEEE, 2005.

[110] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 283–298. Springer, 2006.

[111] Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In *SODA*, volume 98, pages 574–583, 1998.

[112] David M Gillies, Dz-ching Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 114–125. IEEE, 1996.

[113] Alexandre E Eichenberger and Edward S Davidson. Register allocation for predicated code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191. IEEE, 1995.

[114] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[115] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[116] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[117] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2008.

[118] Alkis Evlogimenos. Improvements to linear scan register allocation. *University of Illinois, Urbana-Champaign*, 2004.

[119] Jakob Stoklund Olesen. Greedy register allocation in LLVM 3.0. `http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html`. Accessed: 2023-12-12.

[120] Andreas Fried, Maximilian Stemmer-Grabow, and Julian Wachter. Register allocation for compressed isas in llvm. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 122–132, 2023.

[121] S VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. Rl4real: Reinforcement learning for register allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 133–144, 2023.

[122] Alfred V Hoe, Ravi Sethi, and Jeffrey D Ullman. Compilers—principles, techniques, and tools. 1986.

[123] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[124] Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for ssa-form programs. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5501 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2009.

[125] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Constant-Loop Dominators for Single-Path Code Optimization. In Peter Wägemann, editor, *21th International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, volume 114 of *Open Access Series in Informatics (OASIcs)*, pages 7:1–7:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[126] Daniel Kästner, Reinhard Wilhelm, and Christian Ferdinand. Abstract interpretation in industry–experience and lessons learned. In *International Static Analysis Symposium*, pages 10–27. Springer, 2023.

[127] Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J Cazorla, Philippa Ryan Conmy, Mikel Azkarate-Askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2015.

[128] Emad Jacob Maroun, Martin Schoeberl, and Peter Puschner. Two-step register allocation for implementing single-path code. In *2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 1–12. IEEE, 2024.

[129] Reese T Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, 1959.

[130] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[131] Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut. Cawet: Context-aware worst-case execution time estimation using transformers. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[132] Gabriella Bettonte. *Quantum approaches for Worst-Case Execution-Times analysis of programs*. PhD thesis, Université Paris-Saclay. NNT: 2023UPASG026. tel-04082236, 2023.

[133] Fanqi Meng and Xiaohong Su. Reducing wcet overestimations by correcting errors in loop bound constraints. *Energies*, 10(12):2113, 2017.

[134] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II 6*, pages 493–508. Springer, 2014.

[135] Zbigniew Chamski. *Fast and efficient generation of loop bounds*. PhD thesis, INRIA, 1993.

[136] Jinghao Sun, Nan Guan, Zhishan Guo, Yekai Xue, Jing He, and Guozhen Tan. Calculating worst-case response time bounds for openmp programs with loop structures. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 123–135. IEEE, 2021.

[137] Amir M Ben-Amram and Geoff W Hamilton. Tight worst-case bounds for polynomial loop programs. In *Foundations of Software Science and Computation Structures: 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 22*, pages 80–97. Springer, 2019.

[138] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: An effective technique for vliw and superscalar compilation. In *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*, pages 229–248. Springer, 2011.

[139] Steve Haga, Andrew Webber, Yi Zhang, Nghi Nguyen, and Rajeev Barua. Reducing code size in vliw instruction scheduling. *Journal of Embedded Computing*, 1(3):415–433, 2005.

[140] Vasileios Porpodas and Marcelo Cintra. Caesar: Unified cluster-assignment scheduling and communication reuse for clustered vliw processors. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2013.

[141] Paul Lokuciejewski, Timon Kelter, and Peter Marwedel. Superblock-based source code optimizations for wcet reduction. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1918–1925. IEEE, 2010.

[142] Xuesong Su, Hui Wu, and Jingling Xue. An efficient wcet-aware instruction scheduling and register allocation approach for clustered vliw processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–21, 2017.

[143] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *Acm sigplan notices*, 35(5):121–133, 2000.

[144] Abid Malik. Constraint programming techniques for optimal instruction scheduling. 2008.

[145] Rajiv Gupta. Generalized dominators and post-dominators. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 246–257, 1992.

[146] John L Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):422–448, 1983.

[147] Mathieu Jan, Mihail Asavoae, Martin Schoeberl, and Edward A. Lee. Formal semantics of predictable pipelines: a comparative study. In *TODO: ASP-DAC*, January 2020.

[148] Martin Schoeberl, Bekim Cilku, Daniel Prokesch, and Peter Puschner. Best practice for caching of single-path code. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[149] Emad Jacob Maroun, Eva Dengler, Christian Dietrich, Stefan Hepp, Henriette Herzog, Benedikt Huber, Jens Knoop, Daniel Wiltsche-Prokesch, Peter Puschner, Phillip Raffeck, Martin Schoeberl, Simon Schuster, and Peter Wägemann. The platin multi-target worst-case analysis tool. In *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2024.

[150] Michael Platzer. *Predictable and Performant Computer Architectures for Time-Critical Systems*. PhD thesis, TU Wien, Faculty of Informatics, 2023.