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

Emad Jacob Maroun
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
emad.maroun@tuwien.ac.at

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Kongens Lyngby, Denmark
masca@dtu.dk

Peter Puschner
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
peter@vmars.tuwien.ac.at

***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.**

***Index Terms*—real-time systems, time-predictable computer architecture, single-path code generation.**

I. Introduction

Single-path code allows time-predictable execution of real-time applications [1]. 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 [2], as part of the T-CREST project [3], 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 [4]. 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 [2] 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 [4]; 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 [5]. 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 [4].

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

II. Related Work

The single-path code generation approach is introduced in [6] for use in real-time systems. The authors of [4] 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 [7] 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 [8] and [9] 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 [10], 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 [11]. The authors present a technique for aligning single-path loops with the instruction cache to reduce cache misses during loop execution.

In [12], 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 [13]–[16]. Variations on these techniques can also produce optimal schedules—given enough time and based on some definition of *optimal* [17]. 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 [18]) 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 [19] 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 [20] 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 [21], 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

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

result would increase the single-path code's execution time by 90 cycles, even though the worst-case 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.

III. 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.

A. The Patmos Processor

Patmos [2] is a processor that is part of the multi-core architecture T-CREST [3]. T-CREST and Patmos aim to build time-predictable computers [22], including time-predictable on-chip communication [23] and memory controllers [24]. Patmos itself is a RISC-style processor optimized for real-time systems. It has two in-order pipelines and therefore avoids any timing anomalies [25]. 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 [26], [27]. 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 [28], which is simple to analyze [29]. 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.

B. 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 [30], 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

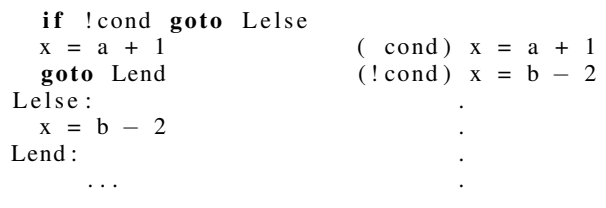

Fig. 1. The difference between branching and predicated execution. On the left, a condition makes the execution skip one of the paths, while on the right, both paths are executed, but only one of them will be enabled at a time.

used as a source for 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 [31]. 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:

1) 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 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.

2) 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.

3) 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 [4], [32]. 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.

C. 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.

IV. 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 [4]. In this section, we will describe the transformation informally and highlight the differences from the original work. Section IV-B is wholly our contribution to the single-path transformation, while the descriptions in subsection IV-A and subsection IV-C are identical to the original work unless where we explicitly state otherwise.

A. 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 2 as an example. For the loop with the header `b`, the transformation will create the FCFG seen in Figure 3. Since the whole procedure is also treated as a loop, the FCFG in Figure 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.

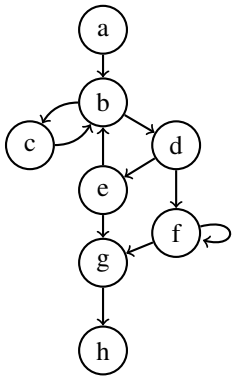

Fig. 2. A control-flow graph of a procedure.

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. Subsection IV-C then continues with the transformation's reordering steps, which are mostly identical to the original work.

B. 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 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 V, 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.

[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.

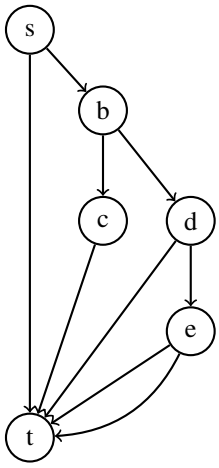

Fig. 3. FCFG of the loop `b`.

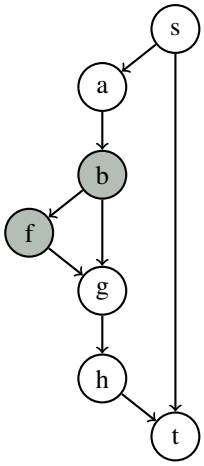

Fig. 4. FCFG of the pseudo loop `a`, with the headers of loops `b` and `f` in grey.

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 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.

C. 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

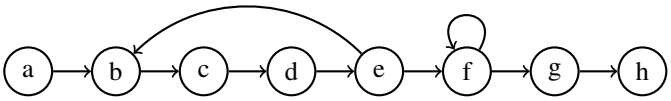

Fig. 5. The single-path CFG transformed from the one in Figure 2 with bundling disabled.

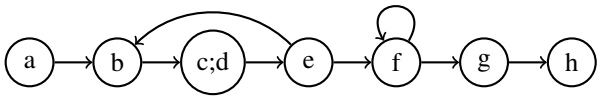

Fig. 6. The single-path CFG transformed from the one in Figure 2 with bundling enabled.

our example procedure—without any bundling—can be seen in Figure 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`.

V. Implementation

We extend the open-source implementation presented in [4] 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.

A. Compiler Overview

The compiler is based on the LLVM compiler framework [33]. 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.

B. 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:

`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 2 will result in the bundling of the blocks `c` and `d`, as can be seen in Figure 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.

VI. 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.

A. 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 I 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.

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.

[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.

TABLE I
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 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.

B. TACLe Benchmarks

We also evaluate our algorithm's performance on 13 programs from the TACLe benchmark suite [34]. 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 I. This is caused by the bundling algorithm's simplicity, which is not sophisticated enough to see bundling opportunities in the remaining programs. In Table II, 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

TABLE II
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 III
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

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 III 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.

C. 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 [35]. 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 I 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 II and Table III, 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 I) using `pasim`'s `--bsize` flag. This causes cache misses to have a lower penalty, as data is loaded faster. As seen in Table IV, 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.

TABLE IV
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

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 [36], 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.[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]

VII. 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 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.

[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.

ACKNOWLEDGMENT

This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien.

REFERENCES

[1] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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 Distributed Computing (ISORC 2017)*, pages 76–83, Toronto, Canada, May 2017. IEEE.

[11] 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.

[12] 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.

[13] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Computer Architecture Letters*, 30(07):478–490, 1981.

[14] 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.

[15] 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.

[16] 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.

[17] Roberto Castañeda Lozano and Christian Schulte. Survey on combinatorial register allocation and instruction scheduling. *ACM Computing Surveys (CSUR)*, 52(3):1–50, 2019.

[18] 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.

[19] Jun Yan and Wei Zhang. A time-predictable VLIW processor and its compiler support. *Real-Time Systems*, 38(1):67–84, 2008.

[20] 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):66, 2016.

[21] 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.

[22] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.