Towards 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*—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.

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

I. Introduction

Patmos [1] has been designed to support the single-path programming paradigm [2]. 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 [3]. 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 [1].

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 [3], 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 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 Patmos 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 [4] for use in real-time systems. The authors of [3] 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 [5] 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 [6].

The authors of [7] 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 [8], 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 [9], 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 [10]. The authors present a technique for aligning single-path loops with the instruction cache to reduce cache misses during loop execution.

Both [11] and [12] 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 [13] 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.

III. 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.

A. The Patmos Processor

Patmos is a RISC style processor optimized for real-time systems [1]. The aim of Patmos, and the whole T-CREST architecture [14], is to build time-predictable computers [15]. It uses an in-order pipeline to avoid any timing anomalies [16]. As the analysis of caches may introduce large overestimations of the WCET, Patmos contains special caches and scratchpad memories to reduce these estimates. Instructions are cached in the so-called method cache [17], [18]. 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 [19].

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.

B. 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.

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

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.

- 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 [20].

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

1) 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 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.

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, 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.

3) 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.

C. 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.

IV. 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 [3]. In this section we will describe the whole transformation informally, highlighting the differences from the original work.

A. 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`.

We use the procedure in Figure 2 as an example. From it, the FCFG generated from the loop with header `b` can be seen in Figure 3. In Figure 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, 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.

B. 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

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

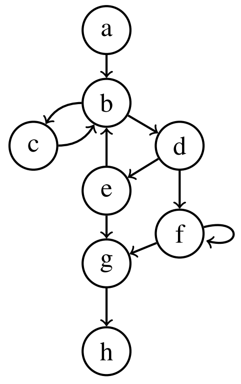


Fig. 2. A control-flow graph of a procedure.

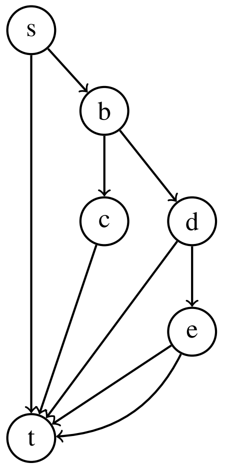


Fig. 3. FCFG of the loop `b`.

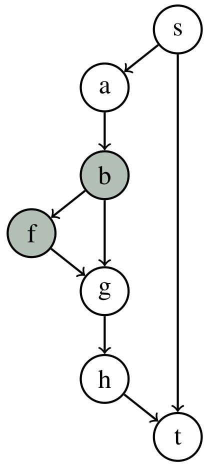


Fig. 4. FCFG of the pseudo loop `a`, with the headers of loops `b` and `f` in grey.

instructions in the 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.
- **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.

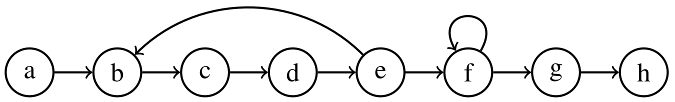
Any bundling algorithm must ensure that any block pairs it bundles adhere to the above rules. It must consist of two parts: (1) `findBlockPair`: Finds a pair of blocks to bundle and (2) `bundleBlockPair`: Does the actual bundling of a pair of blocks, deciding exactly which instructions in each block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

[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.



$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[r] & d \arrow[r] & e \arrow[r] \arrow[lll, bend right] & f \arrow[r] \arrow[loop above] & g \arrow[r] & h \end{tikzcd}$$

Fig. 5. The single-path CFG transformed from the one in Figure 2. No bundling is performed.

C. 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.

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 5 we can see the result of the transformation of the CFG in Figure 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`.

V. Implementation

We build directly on the open-source implementation presented in [3]. 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.

A. Compiler Overview

The Patmos compiler is based on the LLVM compiler framework [21]. 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.

B. 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 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.

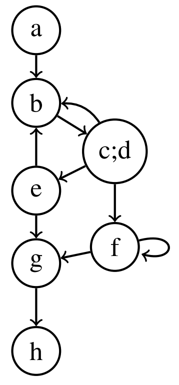


Fig. 6. CFG where `c` and `d` have been bundled.

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

TABLE I
EXECUTION TIME (IN CYCLES) FOR DIFFERENT TACLE BENCHMARK PROGRAMS WITH OR WITHOUT BUNDLING ENABLED. OMITTED THE 5 PROGRAMS THAT PRODUCED NO BUNDLES.

	bsort	countnegative	prime	adpcm_dec
Without bundling	428 418	43 321	24 893	7 173 386
With bundling	428 218	42 439	24 844	7 173 380
Difference	200	882	49	6
Improvement %	0,047	2,036	0,197	0

quence, consecutive ones are merged wherever possible, such that the remaining branches facilitate looping.

VI. EVALUATION

We implemented a simple bundling algorithm to prove the effectiveness of the single-path transformation in supporting any bundling algorithm. To show that the algorithm works and increases performance, we measure the number of cycles used by 9 programs from the TACLe benchmark [22], 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.

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.

Table I shows the number of clock cycles for the execution of each program. 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 II. 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 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 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 in fact increase performance, though, as we expected, not by much. For most

TABLE II
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 III
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

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 [23], 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.

VII. 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

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

therefore needed to more effectively bundle instructions, such that the potential of pairing single-path code with the Patmos architecture can be better realized.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2020.