T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 5.3 Report on Compilation for Time-Predictability

Version 1.0
27 June 2013
Final

Public Distribution

Vienna University of Technology, Technical University of Denmark, AbsInt Angewandte Informatik, University of York

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft João Baptista Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: joao.baptista@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First outline.	15 March 2013
0.2	Sections about stack cache and method cache.	28 May 2013
0.3	Description of single-path transformation.	31 May 2013
0.4	Description of WCET analysis integration and SPM support.	5 June 2013
0.5	Pre-final version, requesting partner comments.	11 June 2013
1.0	Final version	27 June 2013

Executive Summary

This document describes the deliverable *D5.3 Report on Compilation for Time-Predictability* of work package 5 of the T-CREST project, due 21 months after project start as stated in the Description of Work. The deliverable comprises the documentation about compiler technologies used to generate time-predictable code, such as realisation of single-path code generation, patterns for code generation, and explicit control of Patmos components. We present the single-path code transformation algorithm and the function splitter algorithm that have been implemented in the compiler. This report also discusses how the compiler generates code for the time-predictable stack cache and how the tool chain supports the WCET analysis by transforming and generating code annotations.

1 Introduction

For real-time applications, it is important to be able to derive the worst-case execution time (WCET) of the involved tasks in order to be able to analyse the timing behaviour and to guarantee safe operation of the whole system A worst-case analysis must not only consider the timing-relevant properties of the application code itself, but also of the whole platform on which the application is executed. The platform consists of several components: a processor core, memory controller and on-chip memories, but also an operating system and system libraries, all of which affect the execution time. The binary executable that needs to be analysed is generally not written by hand but generated by a compiler.

As discussed in previous reports, features designed to improve the average case performance of traditional hardware platforms impose a number of problems upon the WCET analysis, leading to high pessimism of WCET bounds. Dynamic behaviour of both the software and the hardware in combination with large hardware state spaces leads to a state explosion, making it infeasible to track all possible states of the system under analysis. The timing of operations may depend on the hardware state though. The absence of precise state information therefore makes a precise timing analysis of such operations impossible. In the T-CREST project our goal is to develop a platform that not only enables a WCET analysis to yield tight WCET bounds but also delivers a high performance.

However, it is not sufficient to improve just a single component of existing application platforms in order to achieve both a time-predictability and high performance. Instead, all components of the platform must be designed to be predictable and need to play together to reach that goal. In T-CREST, we therefore envision a platform, where amongst others the compiler, the processor, the memory hierarchy and the WCET analysis are tightly integrated.

In this report we focus on the compiler that generates code for the T-CREST platform and highlight both the code generation strategies as well as the interaction with the Patmos processor, the memory hierarchy and the WCET analysis.

We tackle the problem of the large hardware state space by giving the software explicit control over the hardware state of the processor. This makes the timing of operations far less dependent on the history of previous operations, but requires the compiler to generate code that actually controls the hardware in an efficient way. In turn, this enables new hardware designs that can profit from the high-level view of the compiler, such as the stack cache or the method cache.

As an approach to reduce the dynamic behaviour of the application software, we implemented a single-path code transformation pass in the compiler. This allows the compiler to eliminate input dependent control flow in the application, and thus effectively removes any timing variability of the real-time tasks from normal application code.

Finally, we improve the WCET analysis precision by providing compiler-internal information to the WCET analysis that is lost after code generation by traditional compilers. We developed a set of tools that transform flow information and pass information between the WCET analysis and the compiler.

The rest of the document is structured as following: Section 2 shows how the compiler generates code to explicitly control the Patmos hardware, specifically the method cache and stack cache, the scratch pad, as well as code generation for the VLIW architecture. In Section 3 we present the single-path code generation algorithm that has been implemented into the Patmos compiler. Section 4 describes the integration of the compiler and the WCET analysis. Section 5 contains a list of the requirements

identified in the early project phase, and describes how they are addressed at the current state. We conclude this report in Section 6.

2 Explicit Control of Patmos Components

Generating efficient time-predictable code requires a hardware that is time-predictable and provides mechanisms to control the hardware state, as well as a compiler that emits code that controls the hardware state in a deterministic way. The tight interaction between the code generated by the compiler and the hardware together with the high-level view on the application of the compiler enables more advanced ways of hardware state control, such as caching whole stack frames or function regions.

Precise cache prediction is important for time-predictability due to the high miss penalties, especially on a multi-core setup where the available memory bandwidth is shared by many cores. The Patmos processor provides a number of different, specialised local memories, which enable high memory throughput by making use of efficient DMA-controlled burst transfers, but require support from the compiler. Those local memories are the stack cache (for caching stack frames), the method cache (for caching instructions) and the scratchpad memory (to keep frequently used data local to the processor), as well as an LRU data cache that can be bypassed. The individual local memories have already been presented in previous reports. In the following sections we will describe how the compiler generates code to control those memories.

Furthermore, Patmos features a fully predicated VLIW architecture, for which the compiler must schedule instructions statically. Instead of dynamic branch prediction and out-of-order execution, the compiler performs static scheduling and makes use of speculative execution of code.

2.1 Method Cache

In contrast to traditional instruction caches, the method cache does not manage individual instruction words, but holds entire blocks of code of variable size, e.g., an entire function. The size of these code blocks is limited by the size of the method cache. The compiler thus has to ensure that all code blocks that will be processed by the method cache fit into the cache. In other words, large functions and sometimes even basic blocks within functions have to be split.

We thus implemented a splitting strategy (briefly mentioned in the previous deliverable D5.2 [24]) that exploits graph theoretical properties of the compiler intermediate representation – namely the control-flow graph.

2.1.1 Method Cache Background

The method cache, similar to a standard cache, loads code blocks as needed during the execution of a program. In contrast to a standard cache, however, the cache accesses are guaranteed to be hits except for specific branch (`brcf`), call (`call`) and return (`ret`) instructions. When executing on of these instructions, the method cache is queried whether the code block at the target address is available in the cache. If this is the case, the execution continues without interruption at the beginning of the

target code block within the method cache. In the other case, the method cache starts loading the respective code block (potentially evicting other code blocks) and stalls the processor pipeline until the entire code block has been loaded. The execution then continues as before.

It is important to note that the transfer from one code block into another is only possible to the beginning of the target block, due to the design of the branch and call instructions.

2.1.2 Function Splitting Foundations

The function splitting operates on a compiler intermediate representation called the control-flow graph. Since our approach exploits some graph theoretical properties of this representation we start by giving a few basic definitions:

Control-flow Graphs: The *control-flow graph* (CFG) is a compiler intermediate representation, where nodes in the graph represent so-called *basic blocks* and edges the potential flow of execution of the program at runtime from one basic block to another. A basic block is a sequences of instructions such that whenever the first instruction in the sequence executes all other instructions execute as well, i.e., the sequence does not contain branches except at its very end (accounting for branch delay slots). Furthermore, only the first instruction of as basic block is the target of branches, i.e., when a basic block is executed all its instruction execute. We assume that the CFG of a function has a unique entry point, the so-called root node.

Dominator: A node u in the CFG dominates another node v, when all paths from the CFG's root node to v go through u. Node u is then called a dominator.

Strongly Connected Component: A *strongly connected component* (SCC) is a subset of nodes in the CFG such that a path exists from every node in the subset to every other node in the subset. The CFG can be decomposed into its strongly connected components. We then call an SCC *trivial* when it represents a single node without a self-cycle. All other forms of SCCs are called *non-trivial*.

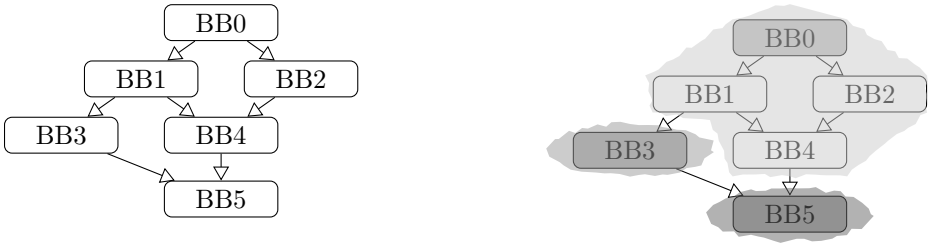
Loop Header: We call a node a *loop header*, when it is part of a non-trivial SCC and a path exists that leads from the CFG's root to the node without passing through any other node in the same SCC. Note that several loop headers may exist within an SCC. An SCC that has a unique loop header is called a *natural loop*, all other forms of SCCs are called *non-natural* loops. A CFG is called *reducible* when all SCCs are natural loops.

Back Edge: A *back edge* is an edge leading from a node within an SCC to one of the SCC's loop headers.

2.1.3 Function Splitting

We can now model the function splitting problem as a partitioning problem on the control-flow graph of a function. The graph has to be partitioned into subsets, that we call *regions*, such that:

- The total code size of each *region* is smaller than the size of the method cache.
- Each region has a distinguished node, the *region header*.
- The source and destination of CFG edges are either both in the same region or the destination node is a region header.



(a) Original CFG

(b) Regions of the CFG

Figure 1: A simple control-flow graph and a possible partitioning into three regions.

Example 1. *Consider, for example, control-flow graph and its partitioning into three regions depicted in Figure 1. The first partition consists of the region header* `BB0` *and basic blocks* `BB1`*,* `BB2`*, and* `BB4`*. The other two regions are singletons, consisting only of their region headers* `BB3` *and* `BB5` *respectively. Note that all control-flow edges from one region into another always lead to the region header of the destination region.*

It follows from these properties that a region header has to dominate all the other nodes within the region.

Lemma 1. *For every valid partitioning of a function's CFG into regions, the region headers dominates all nodes of their respective region.*

Acyclic CFGs: Using this Lemma, we can easily solve the partitioning problem using a simple topological graph traversal on acyclic CFGs. The traversal starts with the CFG's root node, which by definition has to be a region header. During the traversal we grow regions as much as possible by adding the newly visited nodes. Whenever a node is visited, all its predecessors in the CFG have already been assigned to a region. We thus can decide locally whether the node can be added to the region (a region grows) or whether the node has to become a region header on its own. The node can be added to an existing region if:

- All predecessors are member of the same region
- and the region's size does not exceed the size of the method cache when adding the node.

Note that in certain cases additional branches have to be inserted into the program in order to ensure the proper transition between two distinct regions. This bookkeeping can be easily integrated with the approach from above.

Example 2. *Considering a method cache with* 4 *words in size and the acyclic CFG of Figure 1, where all basic blocks are assumed to be* 1 *word in size, the splitting algorithm from above proceeds as follows. The CFG's root node* `BB0` *is visited first, its region thus for now has size* 1*. We then visit* `BB1`*. As it fits into the region of its unique predecessor* `BB0`*, that region grows to size* 2 *by adding* `BB1` *to it. Next,* `BB2` *is visited, which can again be added to the region of* `BB0`*, which by now has grown to size* 3*. The algorithm then visits* `BB4`*, which has two predecessors. Both of them are assigned to the same region. The region furthermore is of size* 3*, which means we still can grow*

the region by adding `BB4` *without exceeding the size of the method cache. However, the region now occupies the entire method cache and cannot grow any further. Consequently,* `BB3` *cannot be added to the region. The basic block is thus marked as the region header of a second region. The last CFG node visited by the algorithm is* `BB5`*. This node has two predecessors in the CFG, both of which are assigned to different regions, it is thus impossible to grow any of the two regions.* `BB5` *thus becomes a region header of a third region.*

Reducible CFGs: The approach from above cannot be applied to cyclic CFGs, since it cannot be assured for nodes within an SCC that all predecessors have been assigned to a region. However, in reducible CFGs the loop headers of all SCCs dominate the nodes within their respective SCCs. This property can be exploited to extend the approach from above:

Lemma 2. *The loop header of an SCC has to be a region header unless all nodes in its SCC are member of the same region.*

We can now extend the approach from before by making case distinctions for loop headers during the traversal. However, before the traversal starts we remove all back edges from the CFG. For regular nodes the traversal remains unchanged as before. For loop headers, we first check whether the predecessors of the loop header are all in the same region. If this is not the case the loop header has to become a region header and the traversal continues as usual. If all predecessors are within the same region we check whether the region can be grown. This time, however, we not only check whether adding the loop header alone results in a valid region. Instead, we check whether all the nodes in the loop header's SCC can be added at once. If this is the case, all nodes of the SCC are added to the region and skipped by when they are visited during the topological traversal.

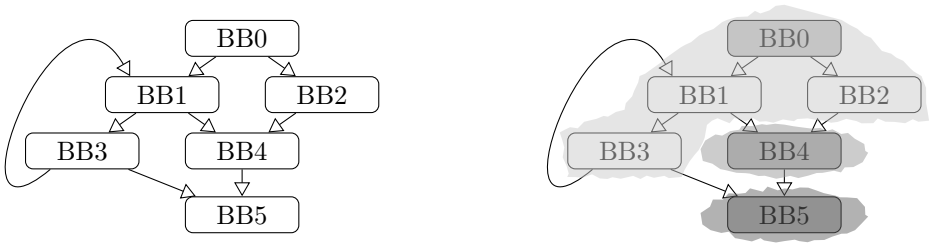
Example 3. *Considering again a method cache of size* 4 *and the reducible CFG of Figure 2, where all basic blocks are assumed to be* 1 *word in size. The CFG is similar to that of the previous example, with the sole difference that a reducible loop exists containing the loop header* `BB1` *and a regular block* `BB3`*.*

In order to apply our algorithm all back edges have to be removed. In this example the only back edge leads from `BB3` *to* `BB1`*. Its removal results in the same CFG as for the original example – where* `BB1` *still remains a loop header for the algorithm.*

The algorithm then processes the nodes of the CFG as before. First, the root node `BB0` *is visited, resulting in the creation of a new region of size* 1*. Then the loop header* `BB1` *is visited. However, instead of simply growing the region by simply adding the node, the whole SCC associated with the node has to be considered, i.e., both* `BB1` *and* `BB3`*. Indeed, growing the region is possible and the two nodes are added to* `BB0`*'s region. The resulting region size is* 3*. Then,* `BB2` *is visited and again added to the region, which by now occupies the whole method cache. Consequently, a new regions are formed when visiting* `BB4` *and* `BB5`*.*

Irreducible CFGs: The approach from above cannot be applied to irreducible CFGs since multiple loop headers may exist for one SCC. However, a lemma similar to Lemma 2 can be stated for loop headers of non-natural loops.

Lemma 3. *The loop headers of an SCC have to be region headers unless all nodes in the SCC are member of the same region.*



(a) Original CFG

(b) Regions of the CFG

Figure 2: A reducible control-flow graph and a possible partitioning into three regions.

The basic idea is to transform the CFG in such a way that all non-natural loops become natural loops. Carrying this transformation out in a way that preserves the program's original semantics, usually leads to an exponential increase in the size of the CFG [7] and would not be applicable in the context of the function splitting problem. We can, however, apply a simpler transformation that only preserves dominance [22]. Once the irreducible CFG has been transformed into a reducible one, we apply the algorithm from above. Note that the transformation is not applied to the program's original CFG but to a copy that is only used for the purpose of computing the partitioning into regions and that is discarded afterward.

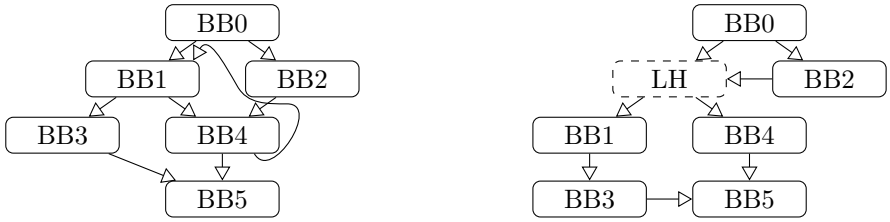
The CFG is iteratively transformed as follows:

1. The CFG is decomposed into its SCCs.
2. All back edges are removed from the graph.
3. For all non-natural SCCs, an artificial loop header is created and all edges leading to/from any of the original loop headers are redirected to this new loop header.
4. This process is repeated until the CFG decomposes into trivial SCCs only.

Example 4. *Figure 3 shows an irreducible CFG along with the transformed graph on which the function splitting operates. The transformation first decomposes the CFG into its strongly connected components, which gives one non-trivial SCC consisting of two nodes* `BB1` *and* `BB4`*. Both nodes are loop headers due to the edges leading to them from* `BB0` *and* `BB2` *respectively. We first remove all the back edges in the graph, which, in this case, are the two edges between* `BB1` *and* `BB4`*. Then an artificial loop header* `LH` *is introduced. All edges that formally lead to the original loop headers of the SCC are redirected to this new node. The resulting graph does not necessarily preserve the original semantics of the program. However, it preserves the original dominance relations (ignoring the new artificial loop headers).*

The resulting graph is then processed as before, considering the new node `LH` *as the SCC's loop header.*

Jump Tables: The approaches described so far assumed that the processing of individual nodes in the CFG can be done independently, accounting only for the special case of SCCs. In practice this is not



(a) Original CFG (b) Transformed CFG

Figure 3: An irreducible control-flow graph and the transformed reducible graph used during function splitting.

always the case. The C `switch` statement, for instance, is often translated to an indirect branch using a jump table. The jump table holds the addresses of the potential branch targets, which are indexed by the argument to the `switch`. The problem here is that the region partitioning is constrained by the content of the branch table. We cannot decide independently for each target whether they should be region headers of not, since this depends on a single instruction – the indirect branch. We can only decide whether all successors of the indirect branch become region headers or none.

We can again solve this problem by a simple transformation of the CFG that exploits the similarity between the constraints imposed by SCCs in the CFG and indirect branches. Both impose a choice between all or nothing. The idea here is to make the dependence between the nodes depending on a single branch instruction explicit by introducing artificial CFG edges that connects all the nodes potentially targeted by an indirect branch, effectively turning these nodes into an SCC. We can then apply the strategy for reducible or irreducible graphs described before.

2.1.4 Future Directions

The function splitting algorithm presented above is a heuristic that may or may not provide optimal solutions. It would be interesting to study the theoretical properties of the problem in more detail and investigate whether simple optimal algorithms can be found. As can be easily verified by the examples given above, the ordering in which the nodes are visited has a large impact on the resulting partitioning into regions. Improved strategies that determine good (or even the best) orderings are needed, in particular, when the worst-case execution time should be optimised while splitting functions.

2.2 Stack Cache

A central feature of the Patmos processor is its stack cache [1], which allows to allocate the stack frame of functions partially or as a whole to a dedicated cache. The cache is organised as a circular buffer that allows to allocate and free space in a first-in/first-out (FIFO) order. The hardware automatically manages spilling and filling data to/from main memory as needed. Our experiments [1, 24] show that a simple allocation strategy already leads to a considerable shift in the number and transfer

volume of data accesses. In some cases as much as 75% of the memory accesses are handled by the stack cache.

In addition to the allocation of data to the stack cache, the placement of instructions to manage the allocated space on the stack cache is relevant. The current model assumes three basic operations on the stack cache [10]: reserving space on the stack cache (`sres`), freeing space on the stack cache (`sfree`), and ensuring the availability of data in the stack cache (`sens`). These instructions are generated automatically by the compiler, and thus are potential candidates for optimisations. In addition, bookkeeping instructions to save and restore the stack cache state before and after a thread or task switch have been added to the instruction set of the Patmos processor. These are intended for system software, such as library and operating system code, and thus are not managed by the compiler. We thus limit the remaining discussion on the three basic operations (`sres`, `sfree`, and `sens`).

The stack cache is mainly used to provide temporary memory space to hold the stack frames of functions. The stack frame is mostly used to hold function-local variables and *spill slots*, which are generated by the compiler's register allocation phase. Since stack frames are associated with a function, they are created during the execution of a program whenever a function is entered and are destroyed whenever the execution returns from the function. The first action thus typically is to create the function's stack frame on the stack cache, while the last action is to free the function's stack frame on the stack cache. While executing a function, it might well be that other functions are called and that these functions similarly allocate space on the stack cache. The corresponding allocations might occupy a large portion of the stack cache and thus cause the stack frames of the calling function to be *spilled* to main memory. When the execution now returns to the calling function, it has to be ensured that the calling function's stack frame is readily available in the stack cache. This is done by placing ensure instructions (`sens`) after every call instruction, since it is not known statically whether executing a call causes spilling or not.

Definition 1. *We denote by* stack cache occupancy *(or short occupancy) the amount of space allocated on the stack cache at the entry of a given function.*

Considering the simple placement strategy of the stack management instructions described in the previous paragraph, it becomes immediately apparent that the occupancy at the entry of a function depends on the calls that finally lead to the invocation of the function. This means that the occupancy depends on the nesting of the program's functions and requires a context-sensitive program analysis. This is particularly relevant when the worst-case spilling behaviour of the reserve instructions of a program needs to be analysed.

Definition 2. *We denote by* stack cache displacement *(or short displacement) the amount of data spilled from the stack cache during the execution of a call.*

In contrast to the stack cache occupancy, the displacement is independent of the nesting of functions that lead to the execution of the call instruction. It only depends on the nesting of the called functions and thus does not require a context-sensitive program analysis.

Example 5. *Consider a stack cache size of 4 words and a simple program with three functions* `A`*,* `B`*, and* `C` *as shown by Figure 4. The corresponding call graph, i.e., a graph representing the relation*

```
(1) func A() {
(2)   sres 2;
(3)   B();
(4)   sens 2;
(5)   C();
(6)   sens 2
(7)   sfree 2;
(8) }
```

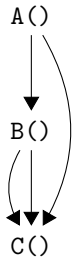
(a) Code of A

```
(1) func B() {
(2)   sres 3;
(3)   C();
(4)   sens 3
(5)   C();
(6)   sens 3
(7)   sfree 3;
(8) }
```

(b) Code of B

```
(1) func C() {
(2)   sres 2;
(3)   sfree 2;
(4) }
```

(c) Code of C



(d) Call Graph

Figure 4: A simple program consisting of three functions and their calling relations.

between functions calling each other, is shown in Figure 4d. Function `C` *is called from three different* contexts. *The first context is* $A \xrightarrow{(3)} B \xrightarrow{(3)} C$*, i.e., function* `A` *calls* `B` *on line 3, which in turn calls* `C` *on line 3. The second context is similar* $A \xrightarrow{(3)} B \xrightarrow{(5)} C$ *and refers to call of* `B` *to* `C` *in line 5 instead of 3. The last context is simply* $A \xrightarrow{(5)} C$*.*

The stack cache occupancy at the entry of `C` *varies for all three context, and can be either* 4*,* 3*, or* 2 *for the three context respectively. Note the difference between first context (calling* `C` *on line 3 of* `B`*) and the second (calling* `C` *on line 5). When the first call executes, a part of* `A`*'s stack data is already evicted to the main memory and* `C`*'s reserve will evict all of* `A`*'s data as well as one word of* `B`*'s stack frame. When the execution reaches the second call all the data of* `A` *remains evicted, while all* 3 *blocks of* `B` *where reloaded by the ensure instruction on line 4.*

The displacement of `C`*, on the other hand, is* 2 *for all three calling contexts.*

2.2.1 Future Directions

When we take a closer look at the previous example (Ex. 5) we notice that not all ensure instructions are needed for the correct execution of the program. The ensure on line 6 of function `A`, for instance, is useless, as we know that `A`'s stack frame is completely loaded back into the stack cache on line 4 and remains present even during the execution of `C`.

We have developed an optimisation that eliminates ensure instructions, which cause most of the overhead (code size, additional execution cycles) related to the stack cache. The key observation to perform such an optimisation is that the amount of data loaded by an ensure can be bounded by examining the maximum stack cache displacement on the program paths between the ensure and its corresponding reserve operation. We will describe this optimisation in deliverable D5.4.

The concepts of stack cache occupancy and stack cache displacement are highly related to the problem of analysing the worst-case behaviour of the stack cache within a WCET analysis tool (such as aiT). We currently work on formulating an efficient analysis using these concepts that can be used for both improved compiler optimisations and better WCET analysis.

2.3 Data Scratchpad Memory (SPM)

A Scratchpad memory (SPM) provides a time-predictable way to store instructions and data [15]. An SPM is a local memory that is closely coupled with a CPU.

SPMs are commonly found within microcontrollers, such as the Atmel AVR [3] or Microchip PIC [16] series, where they may be the only form of RAM. They may also be found within larger CPUs such as Cell [12] where external RAM is also available.

An SPM implements *constant-latency* access to a small amount of private RAM, local to a single CPU. This memory may be shared by all applications running on that CPU. Typical SPM sizes are similar to *level 1* (L1) cache sizes, i.e. 2Kb to 16Kb [18].

2.3.1 Constant Latency

"Constant-latency" means that every access requires the same length of time to serve, regardless of the address [4]. This is unlike a cache, where the latency of each access depends on previously-used addresses. In some respects, SPMs are an ideal memory technology for embedded real-time systems, because any load or store operation targeting the SPM is guaranteed to have a constant execution time [18]. This simplifies WCET analysis, because there is no need to analytically classify each load operation as "hit", "miss", "first miss" and so on [25]. All SPM accesses can be classifieed as "hits".

Typical access latencies are a single CPU clock cycle for loads and stores, as SPMs are typically small and placed near the CPU [25].

2.3.2 Accessing the SPM

In many systems, the SPM is mapped to a specific range of physical memory addresses. Loads and stores using these addresses are routed to the SPM. However, another option is to introduce *typed* loads and stores where a part of each instruction opcode indicates the type of memory to be used. This simplifies WCET analysis because it is easy to determine if an access will be routed to the SPM. Patmos uses typed loads and stores to distinguish between accesses that will be handled by the stack cache, by the data cache and by the SPM.

2.3.3 DMA Controller

Sometimes, an SPM is accompanied by a *direct memory access* (DMA) controller [28]. The purpose of this hardware device is to implement high-speed copy operations between the external memory and the SPM. This is used to load the SPM with code and data as required, and to copy modified data back to the external memory. The same functionality is possible using software (e.g. `memcpy`) but the DMA controller is likely to be faster because it can take full advantage of bus and memory capabilities such as *burst transactions*. Bursts allow more than one word to be sent together, reducing the overhead of large transfers.

Within the Patmos architecture, the data SPM will be paired with a suitable DMA controller. The DMA controller will operate as a co-processor, i.e. in parallel with the CPU. It will be controlled by

program instructions which initiate transfers between SPM and external memory, and which wait for the current transfer to complete. The DMA controller will support a maximum of one transfer at a time, as multiple transfers are not useful in this context.

2.3.4 SPM as a Replacement for Cache

SPMs have disadvantages which have limited their uptake within larger embedded systems.

Each application is forced to take an active role in memory management, statically reserving parts of the SPM for variables [23]. Static allocation is practical for small microcontroller applications (thousands of lines of code) but not for larger applications (millions of lines of code) as an SPM is generally too small to store all of the variables required. In this case, some paging or overlay scheme is required to share the limited space between various parts of an application [8, 9].

The need for a paging scheme is a source of many practical problems. In order to make use of the time-predictability advantage of an SPM, paging cannot be a reactive process that is triggered by loads and stores within the program. Instead, it must occur at predetermined points such as method calls and returns. Some of the major issues are:

1. Determining the best places to carry out paging (*partitioning*) is a complex optimisation problem. It requires detailed information about the program's use of data, and finding optimal solutions may require exhaustive searches. The problem may be simplified by reducing the degrees of available freedom, e.g. only partitioning the program at method call/return boundaries, but this will mean that the SPM and memory bandwidth are not used to the greatest effect. Unfortunately, partitioning techniques do not scale well, or are limited to simplified program representations such as trees [28].

2. It is necessary to carry out some form of analysis to be sure that data is only accessed when it is "paged in", i.e. present in the SPM. This is a difficult problem if dynamic data structures are used [9].

3. Programs have to be modified to page themselves in and out of the SPM. This requires the use of self-modifying code when using an instruction SPM.

4. If multiple applications share the same areas of an SPM, the *real-time operating system* (RTOS) must be able to save and restore the SPM state used by each application [27].

Together, these issues limit the usefulness of an SPM as a cache replacement. The complex WCET analysis required for a cache is replaced by equally complex analysis for the partitioning problem, and the WCET might be increased by the paging process and by inefficiencies in partitioning.

Therefore, the T-CREST project does not solely rely on an SPM. Patmos has a conventional data cache, a method cache, and a stack cache in addition to an SPM. The challenge is to use the SPM alongside these in order to reduce application WCETs.

2.3.5 SPM Alongside Cache

In the Patmos architecture, the main purpose of the SPM is to reduce pressure on the data cache. The method cache provides dedicated local storage for instructions (Section 2.1) while local variables are stored in the stack cache (Section 2.2). The remaining data is stored by default in the data cache, but may be placed in the SPM instead.

The SPM can improve both predictability and performance when leveraged for common code patterns that interact suboptimally with a data cache. The general form of each pattern is an iteration through linear arrays in which some data is loaded or stored, a pattern typically found within matrix transformations, filters, compression algorithms and sort/search algorithms [26]. In each case, the arrays are assumed to be larger than the SPM size. A simple example is:

```
for (i = 0; i < N; i++) {
    load(array[i]);
}
```

but more general forms are possible, for example where multiple arrays are used:

```
for (i = 0; i < N; i++) {
    load(array1[i]);
    load(array2[i]);
    load(array3[i]);
}
```

and where loads and stores are mixed:

```
for (i = 0; i < N; i++) {
    store(array2[i], load(array1[i]) + load(array2[i]));
}
```

and where the step is not constant:

```
for (i = j = 0; i < N; i++, j++) {
    load(array1[j]);
    if (load(array2[i])) j++;
}
```

Using a data cache, each of these patterns will execute quite efficiently. The data cache divides memory space into *blocks*, and once any part of a block is accessed, the whole block is loaded into cache. Therefore, an access to `array1[0]` is likely to fetch `array1[1]` into the cache, speeding subsequent iterations until the end of the block is reached. WCET analysis can account for this behaviour by avoiding the assumption that all loads are misses. However, there is still one miss per block.

An SPM and DMA controller can be used to handle the code patterns even more efficiently by avoiding this blocking. This is done by copying data using the DMA controller while the CPU is

running the code. If this is arranged correctly, the CPU never needs to wait for the DMA operation to complete, as data is always available on time.

Furthermore, the SPM handles store operations more efficiently. For time-predictability reasons, the data cache has a *write-through* policy, meaning that each stored word is immediately committed to external memory. This is not an efficient use of memory bandwidth because the words are written one at a time. The combination of an SPM and a DMA controller allows multiple words to be copied to the external memory in a burst.

Together, these benefits will (1) reduce the memory bandwidth required, because the memory bus is used more efficiently, and (2) reduce the WCET of the application, because blocking time is reduced or eliminated.

2.3.6 Modifying Programs to use SPM

The code patterns listed in section 2.3.5 are accelerated by buffering each load/store operation using the SPM. In this section, the buffering process is described for the following example:

```
for (i = 0; i < N; i++) {
    load(array[i]);
}
```

Assume that the SPM can contain up to `2M` array elements, where `2M` $<$ `N`. Take two areas of SPM space, each of size `M`, and label them `A` and `B`.

The CPU should process area `A` while area `B` is filled with data. Then the roles are exchanged. The CPU processes area `B` while area `A` is filled. The example is rewritten as follows:

```
start DMA copy: array[0..M-1] -> A
for (i = 0; i < N; ) {
    start DMA copy: array[i+M..i+2M-1] -> B
    for (j = 0; (j < M) && (i < N); j++, i++) {
        load(A[i]);
    }
    start DMA copy: array[i+M..i+2M-1] -> A
    for (j = 0; (j < M) && (i < N); j++, i++) {
        load(B[i]);
    }
}
```

The DMA controller only processes one transfer at a time, so each `start DMA` operation waits for the previous transfer to finish. Provided that each DMA copy takes less time than each inner `for` loop, the CPU will never be forced to wait.

There is a minor inefficiency in that some of the data copied into the SPM will not be used (up to `2M-1` elements at the end of the array). This may be avoided by introducing conditions for the `start DMA` operations to bound copies at `array[N-1]`. These conditions are omitted for clarity.

The example generalises for all of the code patterns listed in section 2.3.5. If multiple arrays are involved, the SPM must be further subdivided. If store operations are involved, the DMA copies run in the opposite direction, i.e.:

```
start DMA copy: A -> array[i..i+M-1]
```

Arrays may be accessed by both load and store operations. In this case, DMA copies are required in both directions. These are also required if the store operation does not write to every array element. Otherwise, any skipped elements become undefined when written to the external memory.

2.3.7 Limitations

The scheme described above has two important limitations, which apply irrespective of the code pattern that is used.

Firstly, the presence of store operations leads to the possibility of incorrect code behaviour if arrays overlap in memory. If arrays overlap, multiple copies of a single array element might exist within the SPM, and a store operation will only update one copy. There are elaborate schemes to avoid this *aliasing* problem [25], but they have not been shown to be practical. Ultimately, it must be the programmer's responsibility to avoid the possibility of overlapping when using the SPM and DMA controller.

Secondly, after modifications are applied, the code pattern forms a *critical section* requiring dedicated access to the SPM resource. It is possible to save and restore the SPM state across preemptions, thus avoiding the creation of a critical section, but another elaborate scheme is required to do this [27], again of limited practicality.

2.3.8 SPM Buffer API

The code pattern modifications could be applied by the compiler, acting on the intermediate representation of the application. However, this process would be directed by the programmer, specifying the arrays to be stored in the SPM. Without hints from the programmer, the compiler might allocate SPM space too eagerly, perhaps modifying loops where the SPM is not appropriate, e.g. because the array is very small, or because arrays overlap. It might miss opportunities for the use of the SPM by failing to recognise code patterns.

Given that this degree of programmer involvement is required in any case, the modifications may be implemented using inlined C functions, with the advantage that no compiler modifications are needed.

Programs are modified by including a header file (`"buffer.h"`). Inline functions are then called to (1) initialise each buffer, (2) consume/produce elements, and (3) finalise operations on that buffer:

```
SPM_BFE_Buffer buf;
spm_bfe_init(&buf, array, A, M);
for (i = 0; i < N; i++) {
```

```
    load(spm_bfe_consume(&buf));
}
spm_bfe_finish(&buf);
```

Each buffer is automatically managed by the `spm_*` functions. The programmer need only specify the location `A` and the size `M` of the available SPM space, the source/destination `array`, and provide a handle (`buf`). Transfers from external memory are handled by functions named `spm_bfe_*` (`bfe` = buffer *from* external memory). Transfers to external memory are handled by functions named `spm_bte_*` (buffer *to* external memory).

Within the T-CREST project, a proof-of-concept implementation of this API has already been completed. The API has been used to implement an SPM version of the Merge Sort algorithm. Merge Sort is an example of an *external* sort, meaning that it is suitable for sorting data that is too large to fit in memory. In this case, the unsorted data is too large to fit in the SPM, so it is fetched from external memory via the buffer API.

2.3.9 Address Space Attributes

Since Patmos uses typed loads to access the SPM, the compiler has to know statically for every memory access which memory type will be accessed. Since in T-CREST the SPM is managed by the programmer, it is the programmer's task to tell the compiler about memory accesses to the SPM. We use the `address_space` attribute to mark pointers into the SPM. The `newlib` based `libc` implementation for Patmos defines a `_SPM` macro that can be used to prefix the type of all pointers into the SPM. This is in line with the specification for named address spaces in the Technical Report on Embedded C [11].

Example 6. *Figure 5 shows a sample program that uses the SPM on Patmos. It defines a structure `spm_ptrs_t` that contains three pointers to data structures in the SPM. The `main` function places an instance of that structure at the beginning of the SPM and initializes it. Then the function creates a new data structure `data` in global memory using `malloc` and fills it with some data. Then the function uses the `spm_copy_from_ext` provided by the SPM API to transfer the data from global memory to the SPM, and then calls the custom `spm_merge` function, which merges two input lists into a single output list directly on the SPM. Note that the first argument of the `spm_merge` function is a pointer into the SPM and thus needs to be marked as such. Also note that the programmer only needs to specify at the declaration of a pointer that it uses the SPM, not at every actual memory access, as this is done by the compiler based on the type of the pointers.*

The address space is part of the type of a pointer, therefore the compiler can check that pointers to different memories are not mixed. Otherwise this would lead to incorrect code, as the compiler would then generate loads and stores to the main memory to access data on the SPM, or vice versa.

In addition to the SPM, accessing the global memory without data cache can also be controlled via address spaces. IO mapped devices are also accessed with loads and stores to the SPM, but to support future design changes the `libc` library defines a separate `_IODEV` macro for IO devices. Table 1 gives an overview of the supported address spaces.

```
#include <machine/spm.h>

#define INPUT_ELEMS 10

typedef struct {
  _SPM int *input_A;
  _SPM int *input_B;
  _SPM int *output;
} spm_ptrs_t;

void spm_merge(_SPM spm_ptrs_t *ptrs, size_t size) {
  size_t a = 0, b = 0;
  _SPM int* input_A = ptrs->input_A, *input_B = ptrs->input_B;

  // merge pre-sorted lists input_A and input_B into output list
  for (size_t i = 0; i < 2*size; i++) {
    ptrs->output[i] = (input_A[a] < input_B[b]) ? input_A[a++] : input_B[b++];
  }
}

int main(int argc, char** argv) {
  _SPM spm_ptrs_t *ptrs = SPM_BASE;
  ptrs->input_A = (_SPM int*) SPM_BASE + sizeof(spm_ptrs_t);
  ptrs->input_B = (_SPM int*) SPM_BASE + sizeof(spm_ptrs_t) + INPUT_ELEMS*sizeof(int);
  ptrs->output  = (_SPM int*) SPM_BASE + sizeof(spm_ptrs_t) + 2*INPUT_ELEMS*sizeof(int);

  // produce some values into *data
  int *data = (int*) malloc(2 * INPUT_ELEMS * sizeof(int) );
  int *tmp = data;
  for (int i = 0; i < INPUT_ELEMS; i++) *tmp++ = i * 2;
  for (int i = 0; i < INPUT_ELEMS; i++) *tmp++ = i * 3;

  // copy data to SPM and call spm_merge
  spm_copy_from_ext(ptrs->input_A, data, INPUT_ELEMS*sizeof(int));
  spm_copy_from_ext(ptrs->input_B, data + INPUT_ELEMS, INPUT_ELEMS*sizeof(int));
  spm_merge(ptrs, INPUT_ELEMS);

  return 0;
}
```

Figure 5: A short example program demonstrating the use of SPM pointers

2.4 Code Generation for VLIW

Patmos employs two ALU function units to increase the instruction level parallelism (ILP). Patmos uses a Very Large Instruction Word (VLIW) architecture with no interlocking, where operations are statically assigned to the function units. While this significantly reduces the complexity of a pipeline analysis for the WCET analysis in contrast to dynamic instruction scheduling on an out-of-order architecture, it now falls to the compiler to generate a static schedule that maximises the hardware utilization. Since Patmos does not dynamically check for hazards between most instructions, the compiler must also ensure that the instruction schedule respects the latencies of the individual instructions, as well as the delay slots of control flow instructions and the limitations imposed by the hardware on which operations can be bundled to a single VLIW instruction.

We used the VLIW scheduling support provided by the LLVM 3.2 framework to implement scheduling for Patmos. A pre-register-allocation (pre-RA) scheduling pass orders instructions at machine instruction level so that consecutive instructions honor the hardware restrictions for instruction bundles. A separate pass after register allocation decides on which instructions are actually bundled

Address space	Memory	Macro	Defined in header	Notes
0	Global			Default, uses data cache
1	SPM	`_SPM`	`<machine/spm.h>`	
1	SPM	`_IODEV`	`<machine/patmos.h>`	Memory mapped IO devices
2	Stack cache			Compiler managed only
3	Global	`_UNCACHED`	`<machine/patmos.h>`	Bypasses data cache

Table 1: List of address spaces supported by the Patmos compiler

together, using a hardware model of the Patmos pipeline. Finally, a delay-slot-filler pass ensures that delay slots and instruction latencies are honored correctly by reordering the instructions within basic blocks where possible, or by inserting no-op instructions into delay slots where no instructions are found that can be scheduled.

The compiler also supports bundles in the assembler parser and assembler emitter, i.e., the programmer can bundle instructions manually in inline assembler and assembly files. However, in those cases the programmer is responsible for creating a legal schedule of the instructions himself, as the compiler does not schedule hand-written assembly.

The Patmos compiler generates schedules performing speculative execution by employing an if-conversion pass. If-conversion replaces various if-else constructions in the CFG with a single basic block that executes both branches of the original if-else construct. The code in the branches is predicated with the guard (or its negated form) that represents the evaluated if-condition, disabling the effects of the code that was not executed in the original code. After if-conversion, both branches are thus executed speculatively and only the effects of one branch take effect.

While this leads to a higher number of issued instructions compared to the original code, if-conversion increases the scheduling regions for basic-block oriented instruction schedulers, and reduces the number of branch instructions and thus the number of branch delay slots to fill, potentially leading to schedules that require fewer cycles to execute.

During scheduling, the compiler needs to keep track of data dependencies and latencies between instructions. However, no dependency exists between two instructions if they are guarded with different predicates that can never both be `true` at the same time. Such code typically arises when if-conversion has been performed. We added support to the compiler to detect such cases so that code generated by the if-converter can thus be scheduled more efficiently, as all data dependencies between the if-converted branches arising from the used operand registers can be eliminated.

To support evaluation, it is possible to restrict code generation to use the first issue slot only via the `-mpatmos-disable-vliw` compiler option. Furthermore, we extended the Patmos simulator to generate detailed instruction statistics for both pipelines individually if required.

Future Directions

While the LLVM framework has some support for VLIW scheduling in the current version (at the time of writing), we found that the current scheduling infrastructure of LLVM that separates the

instruction scheduling into several passes leads to sub-optimal schedules and to scheduler implementations that are very hard to maintain. We are therefore currently implementing a new scheduler that integrates delay-slot filling, creation of instruction bundles and latency driven scheduling into a single scheduler pass that uses a standard bottom-up list scheduling strategy.

This new scheduler will not only be able to create better instruction schedules, but also enables us to implement more advanced and new scheduling strategies. We plan to integrate the if-converter and the single-path pass with the instruction scheduler to improve the quality of the generated schedules. Furthermore we plan to investigate into global code scheduling strategies (across basic block boundaries) that optimise the instruction schedule for a low WCET.

3 Single-Path Code Generation

Given a piece of code, we can identify three factors that determine its actual execution time:

- The hardware on which the code is to run
- The sequence of instructions
- The context in which the code is executed (execution history, input values)

In the hardware domain, the architecture of the Patmos processor provides means to make the processor timing independent of the execution context. As for the software, we devise a code-generation strategy that ensures that the sequence of instructions executed during a program execution are insensitive to the values of input variables.

By making the execution time of a piece of code independent from input values, we address one aspect of time-predictability: *stability*. Following the notions defined in D1.1, stability is a measure for the variability of the system timing behaviour. Regarding code execution times [21, 13], a stable system exposes a minimal difference between worst-case and best-case execution time. Hence, one could make reasonable predictions for the execution time of every execution by the knowledge of a single execution.

The idea behind the *single-path transformation* is to generate code that follows the same execution trace for whatever input data is received. The single-path transformation is a code generation strategy that extends the idea of if-conversion [2] to transform branching code into code with a single trace. Input-data dependent branching code is replaced by code that uses predicated instructions to control the semantics of the executed code. Loops are treated in a similar way: Loops with input-data dependent iteration conditions are transformed into loops for which the number of iterations is known at compile time, again using predicated instructions to preserve program semantics.

While the single-path conversion was described by Puschner et al. [20] on a high-level program representation, this document describes the transformation on the language-independent CFG level and its implementation targeting the Patmos processor ISA, which has been integrated into the LLVM compiler backend for Patmos.

3.1 Overview

The single-path transformation is performed late in the code generation process: after register allocation, prologue-epilogue insertion, and control-flow optimisations, and before the final scheduling pass. At this stage, the transformation pass operates on the control-flow graph of a machine function, where the order of the basic blocks in the memory is already defined.

The LLVM framework provides only minimal generic support for predicated instructions. Most important passes, like register allocation and instruction scheduling, are unaware of predication. Performing the single-path conversion at that late point in the code generation phase seems adequate as the control-flow graph remains unmodified subsequently and the pseudo instructions introduced by register allocation (e.g. register copies) are already expanded.

As a consequence, care must be taken that temporary registers and stack slots required as additional storage for predicate registers and newly introduced loop counters are reserved and available during the transformation, and this has to be ensured before register allocation (reserve registers) resp. before prologue-epilogue insertion (reserve stack slots). In the following, we describe the main transformation pass and mention steps that required preparation in an earlier phase explicitly.

3.2 Control Dependence Analysis

The single path transformation is based on a technique called *if-conversion* [2], a technique that transforms control dependencies into data dependence's, by making use of *predicated execution*. A predicated instruction is executed conditionally depending on the value of a Boolean predicate, often referred to as *guard* operand: if the value is true, the instruction executes as expected, otherwise its behaviour is that of a no-op, that is, the hardware state remains unchanged. By means of predicated execution, it is possible to replace control flow by conditional execution, like following code snippet suggests:

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

The branching code would be replaced by conditional execution, such that either the first assignment of x or the second assignment effectively executes, and the other has the effect of a no-op:

```
  cond := ...
  ( cond) x := a + 1
  (!cond) x := b - 2
  ...
```

The CFG allowing two possible execution paths, each containing a different assignment of `x`, is transformed to a CFG with a linear sequence of instructions containing both assignments of `x`, where only one of the two assignments is *enabled* by the guard.

The single-path transformation extends this conversion to whole functions (and programs). Our transformation algorithm is based on the ***RK algorithm*** [17], which converts the CFG of an innermost loop to a linear sequence of predicated basic blocks.

A basic concept of this algorithm is the concept of *control dependence*. Informally, if the branch at the end of basic block BB_X determines whether basic block BB_Y is executed, BB_Y is control dependent on that branch.

Given a CFG $\mathcal{G} = \langle V, E \rangle$, the control dependence function maps basic blocks V to control dependencies C,

$$CD : V \rightarrow 2^C$$

such that for any block $v \in V$, $CD(v)$ is the set of its control dependencies. Each control dependence $c \in C$ is denoted as $\pm w$ with $w \in V$ and $+w$ denoting the true edge (i.e., the edge taken when the branch condition at the end of block w evaluates to true) leaving block w and $-w$ for the false edge:

$$CD(v) \equiv \{\pm w \in C \mid v \text{ is control dependent on } \pm w\}$$

Block v is executed if and only if the branch condition or its negation corresponding to one control dependence $c \in CD(v)$ is satisfied. The control dependence function induces a partitioning on the nodes of the CFG into equivalence classes:

$$v \sim w \quad \Leftrightarrow \quad CD(v) = CD(w)$$

Semantically, if $v \sim w$, then on every execution path v is executed if and only if w is executed. To each of the equivalence classes a unique *predicate* is assigned, and the set of predicates is denoted as P.

Given the control dependence function *CD*, the set of vertices V (basic blocks) of a CFG, and the set of predicates P, the RK algorithm computes two functions R and K that determine (1) which predicate is assigned to each basic block, and (2) where each of the predicates is defined.

Function $R : V \rightarrow P$ assigns to each basic block $v \in V$ a predicate $p \in P$, such that v is enabled if and only if p is true. p is true when a control dependence $c \in CD(v)$ is satisfied. $K : P \rightarrow \text{range}(CD)$ maps each $p \in P$ to the corresponding set of control dependencies.

Example 7. *We illustrate the concepts of CD, R and K on the CFG of function `icrc` given in Figure 6, which is taken from the Malardalen WCET benchmark suite. Table 2 shows the control dependence and the predicate for each basic block. In terms of R and K, for example $R(BB_7) = R(BB_{12}) = p_6$ and $K(p_6) = \{-BB_6\}$.*

The original RK algorithm is applied to innermost loops, whose CFG is acyclic in nature. The steps for converting the CFG to a linear sequence of instructions are as follows:

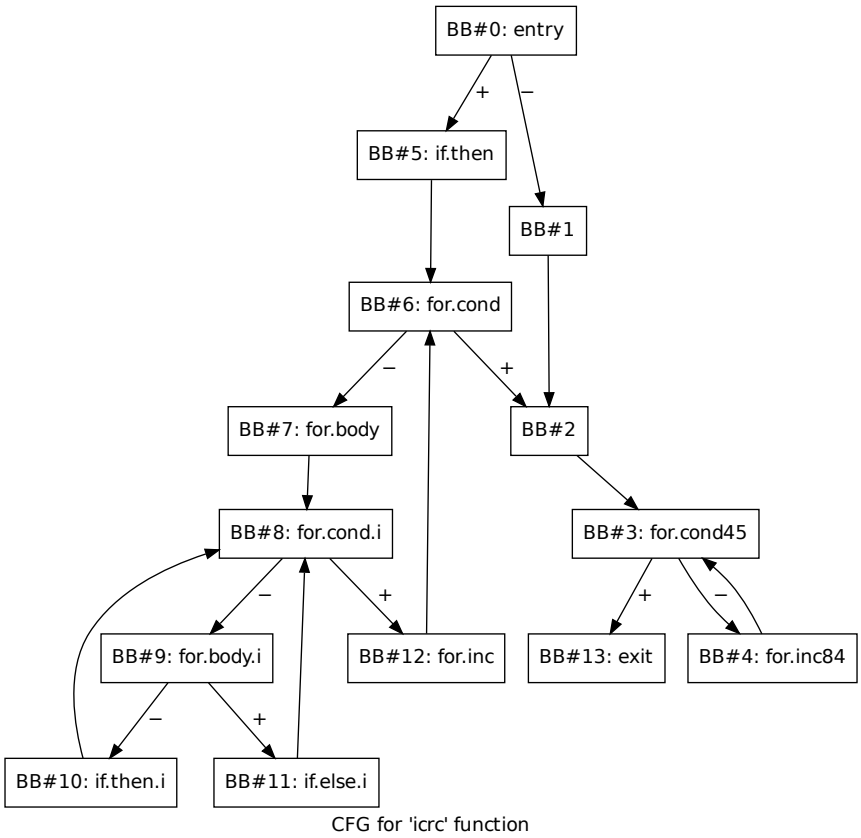


Figure 6: CFG of function `icrc`.

- Compute the control dependence of each basic block in terms of the post dominance relation in a CFG: Given two nodes X and Y, Y is control dependent on X if and only if there is a path in the CFG from X to Y that does not contain the immediate post dominator of X.
- Compute R and K from *CD*.
- Insert predicate assignments according to K. To this end, at each basic block of a given control dependence, the predicate is assigned to the branch condition or is negation: Let $K(p_x) = \{+BB_y, -BB_z\}$, and let cond_y, cond_z be the branch conditions at the end of BB_y and BB_z, respectively. Then, $p_x := \text{cond}_y$ is inserted at the end of BB_y and $p_x := \neg\text{cond}_z$ is inserted at the end of BB_z.
- Guard basic blocks by predicates according to R. A basic block is guarded by a predicate p by assigning p as predicate operand to each of the blocks' instructions. Note that also the assignment statements of the previous step become predicated.
- Make sure that no predicate is uninitialised, by inserting initialisation code of the form $p_x :=$ `false` at the start node, where necessary.
- Lay out the basic blocks in sequence of a topological sort and remove the branch(es) at the end of each block.

Basic block	Control dependence	Predicate
BB_0	$\{\top\}$	p_0
BB_1	$\{-BB_0\}$	p_1
BB_2	$\{\top\}$	p_0
BB_3	$\{\top, -BB_3\}$	p_2
BB_4	$\{-BB_3\}$	p_3
BB_5	$\{+BB_0\}$	p_4
BB_6	$\{+BB_0, -BB_6\}$	p_5
BB_7	$\{-BB_6\}$	p_6
BB_8	$\{-BB_6, -BB_8\}$	p_7
BB_9	$\{-BB_8\}$	p_8
BB_{10}	$\{-BB_9\}$	p_9
BB_{11}	$\{+BB_9\}$	p_{10}
BB_{12}	$\{-BB_6\}$	p_6
BB_{13}	$\{\top\}$	p_0

Table 2: Control dependence and predicates of basic blocks of the function `icrc`. $\top$ denotes control dependence on the function entry.

While the work of Park and Schlansker [17] is motivated by utilising predicated execution for modulo scheduling to speed up the execution of innermost loops, we extend this technique to convert a complete control-flow graph of a function into predicated code with a single instruction trace for all inputs. Hence, we need to address two issues: (1) How to deal with cycles in the CFG, i.e., how (possibly nested) loops are if-converted, and (2) how to adjust the number of iterations for a loop such that the number of iterations is input-data independent. These issues are addressed in the next sections.

3.3 Single-Path Scopes and the Single-Path Scope Tree

We restrict ourselves to loops with a single header and a single successor (or post-exit) node, as usually obtained by structured programs. We can logically collapse loops with their members to their headers recursively, starting from innermost loops.

For the set of basic blocks of a loop, represented by a distinguished header, we introduce an abstraction we refer to as *Single-Path Scope (SPS)*. We also introduce a top-level SPS for the whole function. The blocks within an SPS are sorted with respect to a topological order of the forward CFG, that is, the CFG with the back-edges removed. Each block except the header is possibly a header of a subscope. This way, the CFG of a function is decomposed to a tree, the *Single-Path Scope Tree*, where inner nodes are headers (representing SPSs), and the leaf nodes are ordinary basic blocks. The root SPS is the function with the entry block as header. The left-to-right order of blocks at each level is a fixed topological sort within an SPS (omitting back-edges).

To construct the SPS tree, we use a natural loop analysis. Each SPS stores its parent, the list of member basic blocks and the child SPSs.

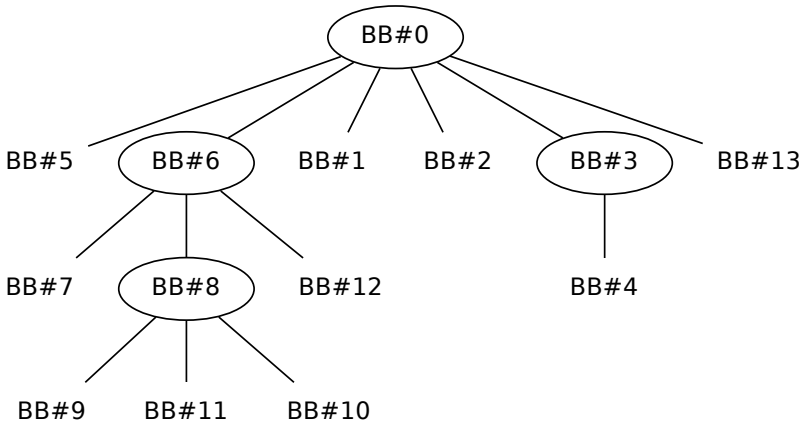


Figure 7: SPS tree of function `icrc`.

Example 8. *Figure 7 depicts the SPS tree of our example* `icrc` *function.* BB_0*,* BB_6*,* BB_8*, and* BB_3 *are SPS headers.*

We can partition the control dependence of a block into two sets: Blocks that are in the same scope, and blocks that are in an outer scope. Consider the control dependence of a header block. It is control dependent on a block from an outer scope, but also on at least one member block, either the header itself or another member block.

Example 9. *The CFG of our example in Figure 6 contains solely such loops with a single header and a single successor. Consider the innermost loop composed of basic blocks* BB_8*,* BB_9*,* BB_{10} *and* BB_{11}*. For* BB_8*,* $R(BB_8) = p_7$ *and* $K(p_7) = \{-BB_6, -BB_8\} = \{-BB_6\} \dot\cup \{-BB_8\}$*.* $-BB_6$ *is a dependence on a block from an outer scope that controls the entry of the loop, and the header* BB_8 *also is control dependent on itself, by* $-BB_8$*: if the branch condition at its end is false, the header will be eventually executed again (iteration). According to the RK Algorithm, definition points of* p_7 *would be described by* $K(p_7)$*: At the end of* BB_6*,* p_7 *is assigned to the negation of the branch condition, which itself is predicated by* $R(BB_6) = p_5$*, i.e.,* $(p_5)\ p_7 := \neg cond_6$*. Similarly for the definition of* p_7 *in* BB_8*, i.e.,* $(p_7)\ p_7 := \neg cond_8$*.*

We compute control dependence and the functions R and K for each SPS separately, which naturally results in the decomposition of control dependence as mentioned above.

Example 10. *Table 3 shows the control dependence and predicate information as computed within each SPS. For example, the execution of the SPS with header* BB_6 *is dependent on* $+BB_0$ *within the top level function (SPS with header* BB_0*), but within the SPS it represents, it is control dependent only on itself, on* $-BB_6$*.*

Basic block	Control dependence	Predicate
$[BB_0]$	$\{\top\}$	p_{00}
BB_5	$\{+BB_0\}$	p_{01}
BB_6	$\{+BB_0\}$	p_{01}
BB_1	$\{-BB_0\}$	p_{02}
BB_2	$\{\top\}$	p_{00}
BB_3	$\{\top\}$	p_{00}
BB_{13}	$\{\top\}$	p_{00}
$[BB_3]$	$\{-BB_3\}$	p_{10}
BB_4	$\{-BB_3\}$	p_{10}
$[BB_6]$	$\{-BB_6\}$	p_{20}
BB_7	$\{-BB_6\}$	p_{20}
BB_8	$\{-BB_6\}$	p_{20}
BB_{12}	$\{-BB_6\}$	p_{20}
$[BB_8]$	$\{-BB_8\}$	p_{30}
BB_9	$\{-BB_8\}$	p_{30}
BB_{11}	$\{+BB_9\}$	p_{31}
BB_{10}	$\{-BB_9\}$	p_{32}

Table 3: Control dependence and predicates of basic blocks of the function `icrc`, within each SPS.

3.4 Single-Path Loops

The actual transformation of the CFG is performed after predicate register allocation. We describe the idea behind the conversion of loops already at this point, for the sake of clarity.

3.4.1 If-conversion of Loops

We extend the predication and linearisation to loops. Obviously, it is not sufficient to apply the RK algorithm directly to the forward CFG, as the resulting sequence of predicated basic blocks would not contain any looping code, and hence could not contain all possible execution paths of the original CFG as sequence of enabled basic blocks.

In the previous section we have seen that the loop headers are control dependent on at least one block that is a loop member (including themselves). This implies that whenever a control dependence of a loop header is satisfied, that header has to be executed again, eventually.

By definition, a loop header node h dominates every loop member m, i.e., every execution path to a loop member m is started with h. Given a natural loop in a CFG, a loop *iteration* is an execution path starting with h and ending in either the source of a back-edge or the source of an exit edge, containing h only once at the beginning. By the abstraction of SPSs, an iteration can be mapped to a sequence of enabled basic blocks within the linear sequence of scope members (assuming that the execution of a sub-loop is captured entirely by the enabling of the corresponding sub-loop header).

An execution path containing multiple loop iterations of the CFG is mapped to a sequence of linearisation sequences, with possibly different enabled blocks in each sequence; the header is enabled in

every iteration. In the last loop iteration, no control dependence of the header block is satisfied in the last iteration.

Example 11. *Consider the loop with header BB_8 of our example (Figure 6). A possible execution path of the function could contain the sequence*
...-BB_7-[BB_8]-BB_9-BB_{10}-[BB_8]-BB_9-BB_{11}-[BB_8]-BB_9-BB_{10}-[BB_8]-BB_{12}...,
*of 4 iterations. This is mapped to a sequence of linearised scope members by following enabling (enabled blocks in **bold**):*
...-**BB_7-[BB_8]-BB_9**-BB_{11}-**BB_{10}-[BB_8]-BB_9-BB_{11}**-BB_{10}-[BB_8]-BB_9-BB_{11}-BB_{10}-BB_{12}**...
In this sequence, the sequence of scope members BB_8-BB_9-BB_{11}-BB_{10} is repeated 4 times, with different enabling each. Termination is caused by setting the predicate given by $R(BB_8)$ to the negation of the branch condition in BB_8, in the last iteration.

In case of termination of the loop, control flow continues at its single successor block. In order to transform a loop to a repetition of a linearised sequence of predicated blocks, we need to insert a new conditional branch at the end of the linearised sequence of member basic blocks, with the header predicate as branch condition.

Because predicate definitions are guarded themselves, we need to ensure that predicates used within a loop are disabled by default for every iteration. To this end, we unconditionally set all predicates used within a loop (except the header predicate) to *false*.

For the transformation of the whole CFG, the linearised sequence of predicated blocks will contain backward branches for loop iterations at the end of each SPS to its SPS header.

Note that if the loop is not executed at all in an execution path, all basic blocks in the linearised CFG, including the header, are disabled.

Example 12. *The linearised sequence of basic blocks in our example (Figure 6) will contain the original basic blocks in following order, where looping is denoted by *:*
BB_0–BB_5–(BB_6–BB_7–(BB_8–BB_9–BB_{11}–BB_{10})*–BB_{12})*–BB_1–BB_2–(BB_3–BB_4)*–BB_{13} .

*To illustrate the concept of a loop on a disabled path, consider the execution path BB_0-BB_1-BB_2-.... This path mapped to following linear sequence of predicated blocks (enabled blocks are **bold**):*
BB_0-BB_5-BB_6-BB_7-BB_8-BB_9-BB_{11}-BB_{10}-BB_{12}-**BB_1-BB_2**-....

3.4.2 Input-data Independent Iteration Count

While the insertion of a backward branch depending on the loop condition results in a semantically correct conversion, the number of iterations is potentially different for each execution path. This is the case when the loop condition is depending on the input data of the program. In a program with a singleton execution path the number of iterations should be equal for all executions. Predicated execution is a suitable method to equalise the number of iterations for input-data dependent loops.

As we have described in the previous section, loop executions are mapped to the repetition of linearised SPS members, with possibly different enablings. Once the loop termination condition holds, the loop header block is disabled and no backward branch to the loop header is performed.

Also, if a loop is on a disabled path, the header is disabled initially. As a consequence, the sequence of loop members is executed without doing any actual computation, as all loop members remain disabled.

Definition 3. *A* disabled iteration *is the execution of the linearised sequence of predicated loop member blocks, in which all blocks are disabled.*

Observation 1. *By appending disabled iterations to any execution of a loop, the program semantics is not altered.*

As we need to know the maximum number of iterations for each loop in a real-time program, we can use this information to equalise the number of loop iterations by means of disabled iterations. Instead of introducing a backward branch that is conditional on the header predicate, we introduce a conditional branch that is conditional on an iteration counter.

Before entering the loop, we load the maximum number of iterations for that loop into that counter, and at the end of the linearised member sequence we decrement the counter by one. If the counter is greater than zero, we branch back to the loop header.

This way, every execution of the loop will have the same number of iterations, of which a suffix possibly consists of disabled iterations.

3.5 Predicate Register Allocation

Computing the predicate information separately for each SPS is also convenient for predicate register allocation. Based on the SPS tree of a function including decomposed control dependence information, the predicates are assigned to physical predicate registers. First, the set of available predicate registers is determined, which is the set of all writable predicate registers excluding the predicate registers used in the machine function. To ensure availability of predicate registers after the general register allocation passes, two of them are reserved for machine functions that are converted to single-path code.

A possible predicate register assignment is computed locally for each SPS. To this end, liveness information is computed for each predicate, and using a simple scan over the sequence of basic blocks, predicate definitions are either assigned to physical registers or locations on the stack, based on the number of available registers. Predicate definitions can be written directly to stack locations, whereas uses must be in a register as they appear as predicate operands in machine instructions. At each use, if no physical register is available, one must be made available by spilling its contents to a stack location. The register is chosen based on a furthest-next-use policy.

The motivation of performing allocation separately for SPSs stems from the structure of control dependencies. On one hand, predicates that are live at the entry of a header node are also live at the exit, with the possible exception of the predicate of the header node itself. On the other hand, liveness of predicates defined in a loop will not extend beyond the scope of the loop. Therefore, at entering a loop, the predicate of a sub-loop header is copied to a new location, such that the original predicate is preserved.

SPS	No. of locations	Assigned physical registers
BB_0	2	$p_{01} \mapsto$ p2, $p_{02} \mapsto$ p3
BB_3	1	$p_{10} \mapsto$ p4
BB_6	1	$p_{20} \mapsto$ p2
BB_8	3	$p_{30} \mapsto$ p3, $p_{31} \mapsto$ p4, $p_{32} \mapsto$ p5

Table 4: Register assignment in SPSs with 5 available physical registers (p2-p6), cf. Table 3. Note that no writable physical register is required for the control dependence on the function entry ($\top$), as p0 is used for this purpose ($p_{00} \mapsto$ p0).

Furthermore, Patmos features the possibility to store the predicate register file as a whole by copying the special purpose register s0. We employ the strategy to store the predicate register file before entering a loop, and restoring it upon leaving it. For this purpose, for each loop nesting level a slot of one byte is reserved on the stack cache.

To minimise the number of stores and restores of the predicate register file, we devised a scheme that avoids unnecessary stores and restores around deeper nested loops, in case there are enough available physical registers for the loop. To this end, the number of required physical locations is added and propagated up the SPS tree through a post-order traversal.

Definition 4. *Let N be an SPS, and N.num be the number of locations required in N. Then, the* cumulative number of locations *N.cum is defined as*

$$\mathtt{N.cum} = \mathtt{N.num} + \max_{\mathtt{C} \in \mathit{children}(\mathtt{N})} \mathtt{C.cum}$$

Given the attributes N.num and N.cum for each SPS N, we can determine whether storing and restoring of the predicate register file (PRF) in s0 can be omitted upon entry and exit respectively, of a given SPS.

Lemma 4. *Let N be an SPS, P its parent SPS and $|P_{avail}|$ the number of available predicate registers. Then, storing/restoring of the PRF at N can be omitted if*

$$\mathtt{P.num} + \mathtt{N.cum} \leq |P_{avail}|$$

Example 13. *Figure 8 shows the SPS tree with the computed attributes. For BB_3 and BB_8, the condition is fulfilled to avoid a store/restore of s0 around them. Table 4 depicts a resulting register assignment. In this example, no stack location is required.*

3.6 Linearisation of the Control-Flow Graph

After computing predicate assignment information, the actual code transformation step is performed.

First, for each SPS, all basic blocks are guarded by predicating the instructions according to the register allocation information. Second, predicate definition instructions are inserted.

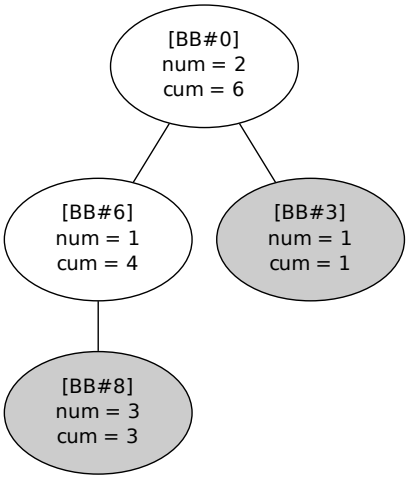


Figure 8: SPS tree headers of function `icrc`, annotated by the attributes of the required number of locations, and the cumulative number of locations. The shaded nodes do not require store/restore of `s0` around them.

Next, the CFG is linearised by a pre-order traversal of the SPS tree. The original branch instructions are removed and the successors of the basic blocks are updated accordingly. Whenever entering a loop, a basic block is inserted, which contains code to store the predicate register file, to copy the header predicate, and to load a loop bound if this information is available. Similarly, whenever going back to a parent node, two basic blocks are inserted: one that contains the backward branch to the loop header, and one after the branch, which restores the PRF. Depending whether loop bound information is available, the loop counter is loaded, decremented, compared against zero, and the comparison result is used as branch condition. If no loop bound information is available, the header predicate is used as branch condition.

As a finalising step, consecutive basic blocks are merged where possible. This provides the subsequent scheduling pass with larger basic blocks.

Example 14. *Figure 9 depicts the CFG of the* `icrc` *function after the linearisation step. Basic blocks* BB_{14}–BB_{20} *are newly introduced blocks around SPSs. Header predicates are copied at loop entries (cf. Table 3):* $p_{20} \leftarrow p_{01}$ *in* BB_{14}, $p_{30} \leftarrow p_{20}$ *in* BB_{15}, *and* $p_{10} \leftarrow p_{00}$ *in* BB_{19}.
Note how the shape of loops has been transformed from while*-style to* repeat-until*-style, as the sequence of blocks is visited at least once. Also, the alternatives at* $BB_0 \rightarrow BB_1/BB_5$ *and* $BB_9 \rightarrow BB_{10}/BB_{11}$ *have been eliminated.*
The CFG after the merging step is shown in Figure 10.

3.7 Future Directions

At this development stage, the single-path code generator is able to convert selected functions. We will extend the functionality to operate interprocedurally, such that all functions reachable from an entry function are also converted.

Although the single-path conversion is able to generate code for loops with a fixed iteration number, the backend lacks support to obtain this information down from higher-level program representations. Efforts in this direction are made to also pass more general flow constraints from bitcode to machine-code.

To optimise the single-path conversion, we will employ a sophisticated input-data dependence analysis to avoid unnecessary transformation of input-data independent control flow.

An obstacle towards completely stable execution times is the access of external main memory. To further improve stability, we plan to make use of the data scratchpad memory (cf. Section 2.3), by restricting memory access to local memories within a task as much as possible.

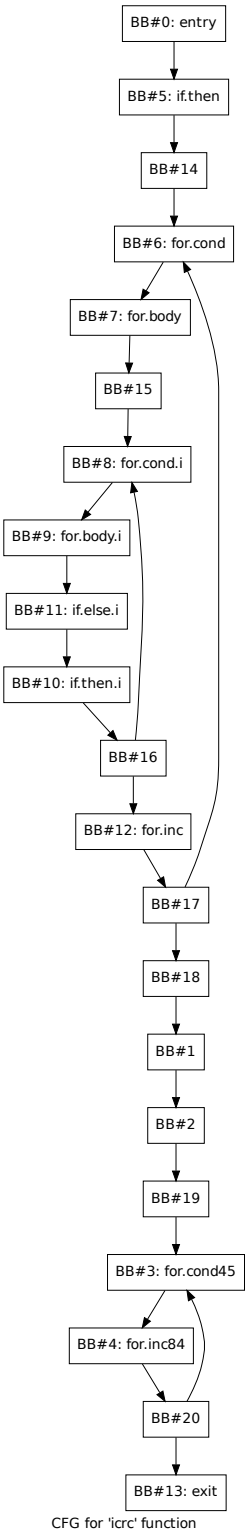


Figure 9: CFG of function `icrc` after linearisation.

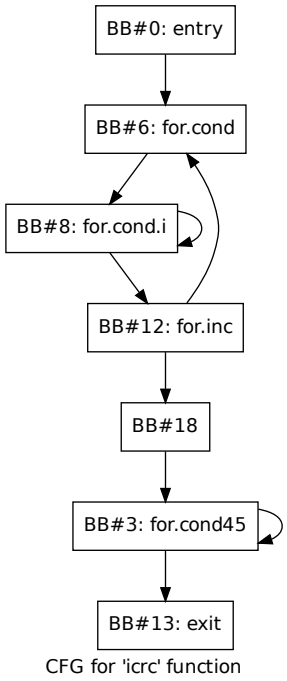


Figure 10: CFG of function `icrc` after merging basic blocks.

4 Integration of Compiler and WCET Analysis

Bernat and Holsti [5] identify four categories of features the compiler should provide to support WCET integration: providing information about the source code's semantics, relating source and machine code, passing information about machine code to the compiler, and control over performed optimisations and machine code generation. As the WCET analysis tool carries out its analysis on binary code, we interpret the first two categories as the challenge to translate information from source code or intermediate code to machine code. This information includes e.g. points-to sets, targets of indirect calls or additional flow information, and stems from both analysis tools operating on higher-level representations, and user annotations. Approximate relations between machine code and source code are made available by existing compilers. While this information is useful for providing feedback to humans, it is not suitable for performing sound WCET analysis.

In addition to the transformation of information between high-level representations and the binary, we are concerned about the exchange of information between compiler and WCET analysis tool at the machine-code level. The machine-code specific information that is crucial to WCET analysis are the structural properties (jump tables, indirect calls) and information about accessed memory locations.

We do not only provide the compiler's knowledge to the WCET analysis tool, but also want to use WCET analysis results to direct optimisations. The WCET analysis tool should thus provide the WCET of program fragments (functions, loops, basic blocks), as well as information on the worst-case path and WCET-criticalities [6].

4.1 Compilation Flow and Preservation of Meta-Information

Figure 11 gives an overview of the compiler tool chain. At the beginning of the compilation process, each C source code file is translated to LLVM intermediate representation (*bitcode*) by the C front-end `clang`. The user application code and static standard and support libraries are linked on this intermediate level by the `llvm-ld` tool, presenting subsequent analysis and optimisation passes as well as the code generation backend a complete view of the whole program. This control-flow graph oriented intermediate representation is particularly suitable for generic target independent optimisations, such as common sub-expression elimination, which are readily available through `llvm-ld`. The `llc` tool constitutes the backend translating LLVM bitcode into machine code for the Patmos ISA, addressing the target-specific features for time predictability. The backend produces a relocatable ELF binary containing symbolic address information, which is processed by `gold`[1], defining the final data and memory layout, and resolving symbol relocations.

Due to the complexity of modern compilers and their optimisations, transforming information from the source level to the machine-code level is not trivial to retrofit into an existing industrial-quality framework such as LLVM. In order to manage the complexity of this problem, we subdivide the transformation of meta-information into a number of steps. We outline this strategy below.

[1] gold is part of the GNU binutils, see `http://sourceware.org/binutils/`

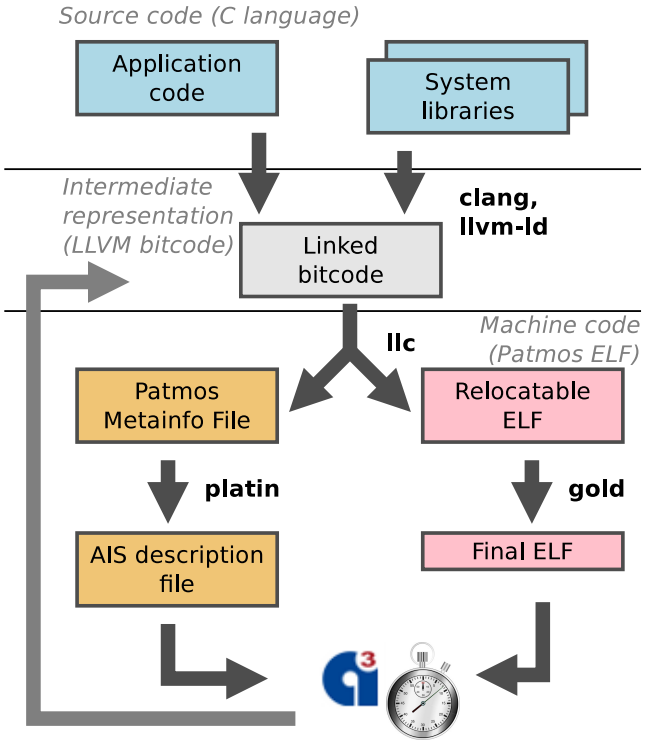


Figure 11: Compiler Tool Chain Overview

From Source Code to Bitcode The translation from source code to the platform-independent intermediate representation, which is performed by `clang`, translates information available only at the source-level (e.g., annotations in form of pragmas) to bitcode meta-information. In order to separate concerns, no optimisations are performed at this stage.

High-Level Optimisations High-level optimisations are performed on bitcode, although the same considerations apply to source-to-source optimisations. Some of the available high-level optimisations perform major structural changes to the program (e.g., loop unswitching). Consequently, these optimisations need to be extended to preserve meta-information which is relevant for timing analysis but not necessarily for the compiler backend. LLVM partly addresses this problem, as it provides infrastructure for bitcode meta-information, and helps to transform or invalidate debug information during optimisations. Techniques for maintaining for example loop bounds, which are crucial for WCET analysis, have been developed [14], but require considerable additional effort for each optimisation. However, as these optimisations are implemented in a platform-independent way, it is likely that the investments on preserving meta-information pay off.

From Bitcode to Machine Code In order to preserve meta-information in the compiler backend, the compiler maintains relations between basic blocks (and memory accesses) at the bitcode and the machine code level. Flow facts are transformed to machine code using these relations. Thus it is not necessary to add dedicated support for flow-fact updates in the backend. Those high-level

optimisations that perform structural changes that cannot be expressed in our relation model, need to be (and are) implemented as bitcode optimisations.

Relating Machine Code and Linked Binaries The relation between relocatable machine code that is emitted by the compiler and the binary executable generated by the linker is simplified by two measures: first, global optimisations are carried out on the bitcode level, but not on the machine-code level. Second, the compiler and linker ensure that all symbolic names necessary to specify information for the timing analysis tool are generated and stored in the binary. This permits all information about machine code to be specified without referring to addresses.

4.2 Information Exchange Language and the Platin Tool Kit

At the core of the information exchange strategy is the *Patmos Metainfo Language* (PML) file format, and the associated tool chain to extract, import and transform information about the program related to WCET analysis.

PML is designed to allow information exchange with different tools at both the bitcode and machine code level. Fundamental concepts such as control-flow graphs, loop nest trees or linear flow constraints are thus defined in a way which is applicable to both bitcode and machine code. The relation between optimised bitcode and machine code is also stored, and allows to transform information obtained from analysis tools operating at the bitcode level. At the machine code level, the PML format attempts to be largely platform independent. To this end, common machine-code related concepts, such as jump tables, can be specified in a uniform way.

In order to simplify adoptions to new target architectures, we integrate a platform-agnostic framework for exporting information about bitcode and machine code into LLVM, so that particular target backends only need to provide some target-specific information, and may extend the exporter's functionality as needed. The integration with `aiT`, and other analysis tools, is realised by a set of routines to parse, transform, merge and generate program information, packaged in the `platin` (Portable Llvm Annotation and TImiNg) tool kit.

Support for emitting the control-flow structure of both the bitcode and machine code, as well as the relation between them is present in `llc`. The resulting PML database is then subsequently manipulated by the tools of the `platin` tool kit.

The `platin` tool kit also includes routines for merging information from different PML files to improve modularity, and a visualisation tool to present control-flow information to humans.

4.3 AbsInt aiT Integration

For timing analysis with the `aiT` tool, an AIS annotation file is exported from the PML file, which in conjunction with the final executable serves as input to the analyser. When the WCET analysis is complete, the analysis results are merged back into the PML database. The WCET analysis results can then be used by the compiler in a second iteration to guide optimisations for improving timing predictability and worst-case performance.

For aiT to be able to correctly reconstruct the control flow from the binary, targets of computed calls and branches must be known. aiT can find many of these targets automatically for code compiled from C. This is done by identifying and interpreting switch tables and static arrays of function pointers. Yet dynamic use of function pointers cannot be tracked by aiT, and hand-written assembler code in library functions often contains difficult computed branches. Targets for computed calls and branches that are not found by aiT must be specified by providing a parameter file called AIS file to aiT.

Traditionally, this parameter file is written by the developer. However, this is a tedious and error-prune task. In T-CREST, the compiler exports its internal information about targets for computed calls and branches to PML. The platin tool kit generates the AIS parameter file, which contains the branch target information. The WCET analysis can therefore profit from the internal high-level knowledge of the compiler about the control flow of the program. This enables an automatic WCET analysis using aiT without additional user interference even in the presence of computed calls and branches.

We also integrated the Patmos simulator into the platin tool kit, which enables us to extract information from simulation traces. Among other useful applications, flow information extracted from simulator traces comes in handy to get started with timing analysis: Where otherwise the aiT analysis tool would not be able to perform analysis due to the lack of loop bounds, the iteration counts extracted from the trace are provided to the tool. This allows the programmer to get an initial (though potentially unsafe) estimate on the timing behaviour of the application in early development stages.

4.4 Example

In this section, we present an example that (1) demonstrates the integration of the compiler and the WCET analysis tool, and (2) hints at the versatility of the platin tool kit.

Base64 is an encoding scheme to represent binary data in an ASCII string format by translating it into a radix-64 representation. Therefore, Base64 encoding converts 3 octets into 4 encoded characters. Figure 12 depicts the C source code of a simplified version of the decoding function. Depending on the current position within four read characters, binary data is written with different bit offsets to the target buffer. This behaviour is realised with a state-machine implemented as switch statement.

The switch statement eventually gets translated to an indirect branch facilitating a jump table, Figure 13. Note that due to the lack of a default case, no switch table size check is generated by the compiler However, instead of putting the burden on the analysis tool to reconstruct the possible branch targets from the object code, which is not always possible anyway, this information is readily available and contained in the PML database that is emitted additionally to the binary by the back-end. Figure 14 depicts the machine control-flow graph as contained in the PML database, where the branch targets are known to be basic blocks 4, 5, 6, or 7.

In order to be WCET-analysable, an upper bound of the while-loop needs to be known statically. In practice, the programmer would annotate the loop bound either on assembly level, a procedure which is tedious and error prone, or, as desired in the T-CREST project, on the source code where the annotation would be co-transformed during compilation.

Lacking an implementation thereof at the current state, we use the opportunity to demonstrate the benefit of the integration of the Patmos simulator `pasim` into the `platin` tool kit, as illustrated in Figure 15: We simulate the decoding function with a string of maximally allowed length as input data and let the trace-analysis tool extract the loop iteration counts for the executed loops, which are annotated back to the PML database. Finally, an AIS annotation file is exported from the latter, containing the relevant annotations enabling WCET-analysis, as shown in Figure 16.

4.5 Future Directions

Support for user-provided flow annotations above the bitcode intermediate level is not yet implemented in the compiler. It involves extending the `clang` frontend to correctly translate flow annotations (such as loop bounds) in the form of C pragmas from the AST-oriented representation to the CFG-oriented bitcode representation, and extending selected relevant CFG-manipulating transformations at bitcode level (*e.g.*loop unswitching) to provide information that allows the `platin` tool kit to co-transform the flow annotations.

We are currently investigating methods to add support for co-transformation of flow annotations from source code level to the LLVM framework and plan to implement them into our compiler in the near future. We also plan to further tighten the integration of the compiler and the WCET analysis by exporting additional compiler knowledge such as memory locations to PML and hence to `aiT`.

The implementation of compiler optimisation passes that use the back-annotated WCET analysis results to optimise specifically for worst-case performance are planned for the upcoming project phase and Deliverable D5.4.

```
const char Base64[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
          "abcdefghijklmnopqrstuvwxyz0123456789+/";
const char Pad64 = '=';

int b64_pton(char const *src, char *target, size_t targsize)
{
  int tarindex=0, state=0;
  char *pos, ch;

  while ((ch = *src++) != '\0') {
    if (ch == Pad64) break;

    pos = strchr(Base64, ch);
    switch (state) {
      case 0:
        target[tarindex] = (pos - Base64) << 2;
        state = 1;
        break;
      case 1:
        target[tarindex]   |=  (pos - Base64) >> 4;
        target[tarindex+1]  = ((pos - Base64) & 0x0f) << 4 ;
        tarindex++;
        state = 2;
        break;
      case 2:
        target[tarindex]   |=  (pos - Base64) >> 2;
        target[tarindex+1]  = ((pos - Base64) & 0x03) << 6;
        tarindex++;
        state = 3;
        break;
      case 3:
        target[tarindex] |= (pos - Base64);
        tarindex++;
        state = 0;
        break;
      default:
        __builtin_unreachable();
    }
  }
  return (tarindex);
}
```

Figure 12: Base64 decoding function. The switch statement with unreachable default branch is translated to a jump table.

```
.LBB2_3:
  16b0: 87 c2 10 0d 00 01 92 28    shadd2  $r1 = $r1, 102952
  16b8: 02 82 11 00                lwc     $r1 = [$r1]
  16bc: 00 40 00 00                nop
* 16c0: 07 00 10 01                br      $r1
  16c4: 00 40 00 00                nop
  16c8: 00 40 00 00                nop
```

Figure 13: The assembly generated for `b64_pton` contains an indirect branch (marked with $*$). No switch table size check is performed.

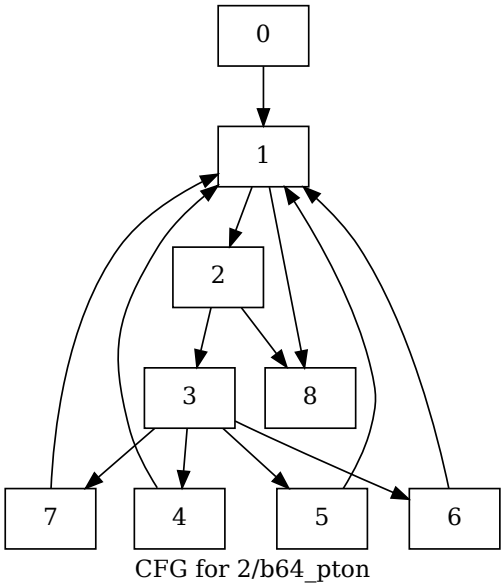


Figure 14: Machine-code level control-flow graph of the `b64_pton` function of Figure 12.

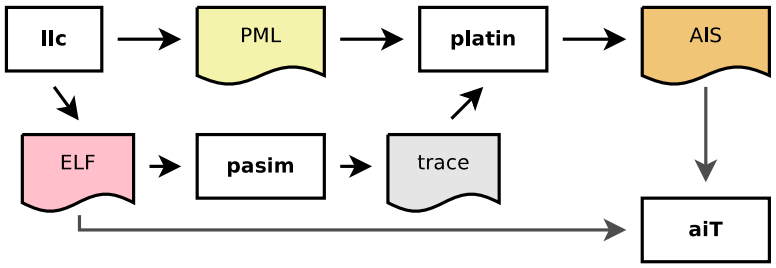


Figure 15: Flow diagram when using `platin` to extract loop bounds of a simulation trace.

```
instruction ".LBB2_3" + 16 bytes branches to
    ".LBB2_4", ".LBB2_5", ".LBB2_6", ".LBB2_7";
    # jumptable (source: llvm)
...
loop ".LBB2_1" max 95 ;
    # local loop header bound (source: trace)
```

Figure 16: The AIS annotations exported from the PML database for the `b64_pton` function.

5 Requirements

We now list the requirements from Deliverable D1.1 that target the compiler work package (WP5) and explain how they are met by the current version of the tool chain or how they will be addressed in the second half of the project period. NON-CORE and FAR requirements are not listed here.

Items for which there has been progress since D5.2 are highlighted in bold font.

5.1 Industrial Requirements

P-0-505 The platform shall provide means to implement preemption of running threads. These means shall allow an operating system to suspend a running thread immediately and make the related CPU available to another thread.

The compiler supports inline assembly, which can be used to implement storing and restoring threads. Further support will depend on the details of the implementation of preemption in the Patmos architecture, developed in the scope of interrupt virtualisation by our project partners (Task 2.6). There is no specific task devoted to the integration of preemption for TUV. We will consider this action during use-case integration (Task 7.2). ***Though, we adapted the ISA to allow storing and restoring the contents of the stack cache to and from the external memory.***

P-0-506 The platform shall provide means to implement priority-preemptive scheduling (CPU-local, no migration).

The compiler supports inline assembly, which can be used to implement storing and restoring threads. Further support will depend on the details of the implementation of preemption in the Patmos architecture, developed in the scope of interrupt virtualisation by our project partners (Task 2.6). There is no specific task devoted to the integration of preemption for TUV. We will consider this action during use-case integration (Task 7.2). ***Though, we adapted the ISA to allow storing and restoring the contents of the stack cache to and from the external memory.***

C-0-513 The compiler shall provide means for different optimisation strategies that can be selected by the user, *e.g.*: instruction re-ordering, inlining, data flow optimisation, loop optimisation.

In the LLVM framework, optimisations are implemented as transformation passes. The LLVM framework provides options to individually enable each transformation pass, as well as options to select common optimisation levels which enable sets of transformation passes.

C-0-514 The compiler shall provide a front-end for C.

The `clang` compiler provides a front-end for C. The compiler has been adapted to provide support for language features such as variadic arguments and floating point operations on Patmos.

C-0-515 The compile chain shall provide a tool to define the memory layout.

The tool chain uses `gold` to link and relocate the executable. The `gold` tool supports linker scripts, which can be used to define the memory layout.

S-0-519 The platform shall contain language support libraries for the chosen language.

The `newlib` library has been adopted for the Patmos platform, which provides a standard ANSI C library.

A-0-521 The analysis tool shall allow defining assumptions, under which a lower bound can be found, *i.e.*a bound that is smaller than the strict upper bound, but still guaranteed to be $>=$ $WCET$ as long as the assumptions are true (*e.g.*instructions in one path or data used in that path fit into the cache).

Extending the `clang` compiler to support flow annotations in the source code is considered in Task 5.3. The annotations will be passed down to the WCET analysis by the tool chain. Flow annotations can be used to add additional flow constraints to the WCET analysis.

S-0-522 Platform and tool chain shall provide means that significantly reduce execution time (*e.g.*: cache, scratchpad, instruction reordering).

The LLVM framework provides several standard optimisations targeting execution time, such as inlining or loop unrolling. ***The data scratchpad memory can be accessed with dedicated macros, which allow the programmer to manually utilize this hardware feature. Instruction reordering is performed statically at compile-time to reduce the number of stall cycles in the processor. The stack cache provides a fast local memory to reduce the pressure on the data cache and to obtain a better WCET bound.*** *Task 5.3 is dedicated to the development of optimisations that focus on reducing the WCET.*

P-0-528 The tool chain shall provide a scratchpad control interface (*e.g.*: annotations) that allows managing data in the scratchpad at design time.

The SPM API has been integrated into the tool chain, which contains both low-level and high-level functions that allow copying data between SPM and external memory and to use the SPM as buffer for predictable data processing, respectively. Accessing data items on the SPM is possible with dedicated macros that use the address space attribute, which is translated to memory access instructions with the proper type in the compiler backend.

C-0-530 The compiler may reorder instructions to optimise high-level code to reduce execution time.

Instructions are statically reordered to make use of delay slots and the second pipeline, and to minimise stalls during memory accesses.

C-0-531 The compiler shall allow for enabling and disabling optimisations (through *e.g.*: annotations or command line switches).

In the LLVM framework optimisations are implemented as transformation passes. The LLVM framework provides options to individually enable each transformation pass, as well as options to select common optimisation levels which enable sets of transformation passes.

C-0-539 The compiler shall provide mechanisms (*e.g.*: annotations) to mark data as cachable or uncachable.

Variables marked with the `__UNCACHED` macro are compiled using the the cache bypass instructions provided by Patmos to access main memory without using the data cache.

S-0-541 There shall be a user manual for the tool chain.

All tool chain source repositories contain a `README.patmos` *file, which explains how to build and use the tools provided by the repository.* ***Additional documentation of the tool chain can be found in the patmos-misc repository.*** *Further information about the LLVM compiler can be found in the LLVM user guide.*[2]

5.2 Technology Requirements

C-2-013 The compiler shall emit the necessary control instructions for the manual control of the stack cache.

The compiler emits stack control instructions to control the stack cache, so that data can be allocated in the stack cache.

C-4-017 The compiler shall be able to generate the different variants of load and store instructions according to their storage type used to hold the variable being accessed.

The compiler backend supports all variants of load and store instructions that are currently defined by the Patmos ISA at the time of writing. ***Support for annotations to select the memory type for memory accesses is provided by dedicated macros.***

C-4-018 The storage type may be implemented by compiler-pragmas.

Support for annotations to select the memory type for memory accesses is provided by dedicated macros.

C-5-027 The compiler shall be able to compile C code.

The `clang` *compiler provides a front-end for C. The compiler has been adapted to provide support for language features such as variadic arguments and floating point operations on Patmos.*

C-5-028 The compiler shall be able to generate code that uses the special hardware features provided by Patmos, such as the stack cache and the dual-issue pipeline.

The compiler uses special optimisations to generate code that uses the method cache, the stack cache and the dual-issue pipeline. Further optimisations that use the features provided by Patmos to reduce the WCET are implemented in Task 5.3.

C-5-029 The compiler shall be able to generate code that uses only a subset of the hardware features provided by Patmos.

All code generation passes that optimise code for the Patmos architecture, such as stack cache allocation and function splitting for the method cache, provide options to disable the optimisations and thus emit code that do not use the special hardware features of Patmos. Future optimisations will also provide options to disable the use of Patmos specific features.

C-5-030 The compiler shall support adding data and control flow information (*i.e.*: flow facts) to the code, *e.g.*: in form of annotations.

Extending the compiler to support flow annotations in the source code is considered in Task 5.3. The annotations will be passed down to the WCET analysis by the tool chain.

[2] `http://llvm.org/docs/`

C-5-031 The compiler shall provide information about potential targets of indirect function calls and indirect branches to the static analysis tool.

The compiler emits internal information such as targets of indirect jumps for jump tables. The tool chain provides means to transform this information to the input format of the WCET analysis tool. Passing down additional annotations provided by the user (e.g. flow constraints) is part of ongoing work (Task 5.3).

C-5-032 The compiler shall pass available flow facts to the static analysis tool.

The compiler emits internal information such as targets of indirect jumps for jump tables. The tool chain provides means to transform this information to the input format of the WCET analysis tool. Passing down additional annotations provided by the user (e.g. flow constraints) is part of ongoing work (Task 5.3).

6 Conclusion

In this document we presented techniques to generate time-predictable code for Patmos.

The method cache is a specialised instruction cache, which provides the property that cache misses only occur at certain instructions. We described the algorithm for splitting functions into smaller cachable regions, which is required in order to be able to compile large functions for Patmos.

The stack cache as a central component of Patmos requires support from the compiler, as control instructions have to be inserted explicitly. We presented two optimisations, for reducing both the number of stack cache instructions and the amount of data on the cache that potentially has to be loaded back from memory.

We showed how the local data scratchpad memory can be utilised for code patterns that interact subobtimally with a data cache. This way, pressure is lowered from the data cache and the reduction of interferences allows both for better utilisation of the data cache and a more precise cache analysis.

The compiler backend has been extended to support bundling and scheduling of instructions for both Patmos pipelines, but additional work in form of an integrated scheduling, bundling and delay slot filling pass is required to achieve a performance increase by the dual-issue feature.

The single-path code transformation is a transformation performed by the compiler that produces code that exhibits a single dynamic program path, for different input data. It is a means to provide execution times with little variability for tasks. We described the transformation steps along with an illustrative example.

We explained how we improve the analysability of the compiler-produced code by building an infrastucture to tightly integrate the compilation and analysis phases. Information available in the compiler is exported to an external file, which is processed by the `platin` tool kit. Among other uses, this information is transformed to the input format of the `aiT` timing analysis tool, to enable timing analysis to produce more precise results.

The upcoming deliverable D5.4 will discuss compiler optimisations that target the WCET of the real-time tasks.

References

[1] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A time-predictable stack cache. In *Proceedings of the Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, SEUS'13, 2013.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 177–189. ACM, 1983.

[3] Atmel Corporation. 8161D-AVR-10/09: 8-bit AVR Microcontrollers. Technical report, 2009.

[4] Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc. CODES*, pages 73–78, 2002.

[5] Guillem Bernat and Niklas Holsti. Compiler support for WCET analysis: a wish list. In *WCET*, pages 65–69, 2003.

[6] Florian Brandner, Stefan Hepp, and Alexander Jordan. Static profiling of the worst-case in real-time programs. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 101–110, New York, NY, USA, 2012. ACM.

[7] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 106–114, New York, NY, USA, 2003. ACM.

[8] Jean-Francois Deverge and Isabelle Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *Proc. ECRTS*, pages 179–190, 2007.

[9] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1:521–540, December 2005.

[10] DTU. D 2.1 software simulator of patmos. Technical report, T-CREST, 2012.

[11] ISO. *ISO/IEC DTR 18037: Extensions for the programming language C to support embedded processors*. International Organization for Standardization, 2003.

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. R&D*, 49(4/5):589–604, 2005.

[13] Raimund Kirner and Peter Puschner. Time-predictable computing. In *Proc. 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 23–34, 2010.

[14] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1–2):72–105, June 2010.

[15] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., 2006.

[16] Microchip Corporation. DS39632C: Microchip PIC18F2455 Data Sheet. Technical report, 2006.

[17] Joseph C.H. Park and Mike Schlansker. On predicated execution. Technical report, Hewlett Peckard Software and Systems Laboratory, May 1991.

[18] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, 2007.

[19] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[20] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In *Proc. SAFECOMP 2012 Workshops (LNCS 7613)*, pages 382–391. Springer, 2012.

[21] Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time software. In *Proc. 1st International Workshop on Software Technologies for Future Dependable Distributed Systems*, pages 1–5, 2009.

[22] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.

[23] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, 2005.

[24] TUV/DTU. D 5.2 initial compiler version. Technical report, T-CREST, 2012.

[25] Jack Whitham and Neil Audsley. Implementing Time-Predictable Load and Store Operations. In *Proc. EMSOFT*, pages 265–274, 2009.

[26] Jack Whitham and Neil Audsley. Investigating average versus worst-case timing behavior of data caches and data scratchpads. In *Proc. ECRTS*, pages 165–174, 2010.

[27] Jack Whitham and Neil Audsley. Explicit Reservation of Local Memory in a Predictable, Preemptive Multitasking Real-time System. In *Proc. RTAS*, pages 3–12, 2012.

[28] Jack Whitham and Neil Audsley. Optimal Program Partitioning for Predictable Performance. In *Proc. ECRTS*, pages 122–131, 2012.