T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 5.2 Initial Compiler Version

Version 1.0
18 September 2012
Final

Public Distribution

Vienna University of Technology, Technical University of Denmark

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.	6 August 2012
0.2	Introduction and Overview.	13 August 2012
0.3	Compiler Features.	15 August 2012
0.4	Evaluation section added.	17 August 2012
0.5	Chapter on usage of compiler.	29 August 2012
0.6	Quality assurance.	3 September 2012
0.7	Requirements finished.	4 September 2012
0.8	Pre-final version, requesting partner comments.	7 September 2012
1.0	Final version	18 September 2012

Executive Summary

This document describes the deliverable *D5.2 Initial Compiler Version* of work package 5 of the T-CREST project, due 12 months after project start as stated in the Description of Work. The deliverable comprises the adaptation of the LLVM framework for the Patmos processor, providing support of single-path code generation. This report presents the design, implementation, and evaluation of a compilation framework for the Patmos processor. Generating executable machine code required the adaptation of complementary components like standard libraries and an object code linker. The code-generator backend has been developed to make use of the full Patmos ISA, including predication support for single-path code generation.

1 Introduction

The compiler is an essential component within the T-CREST project allowing us to implement and run large programs, e.g., applications provided as use-cases by our project partners **GMVIS Skysoft** and **Intecs**, on the time-predictable Patmos processor. This work package covers the entire tool chain required to translate an application from its source code to the final machine code that can be executed by the processor. The tool chain thus naturally includes the compiler itself, but in addition needs to cover standard libraries and utilities such as an assembler and a linker.

To comply with the requirements of the deliverable, i.e., providing an initial compiler version, hence demanded a series of contributions. A compiler backend has been developed to emit assembly and object code, supporting the complete fully-predicated instruction set of Patmos. An assembler and a disassembler have been adapted. A linker has been extended to target Patmos object code. Standard libraries have been ported. All steps went along with continuous building and testing, which is why this processes have been automated.

The compiler tool chain is centred on the LLVM compiler framework [7], an open-source collection of tools and libraries to implement compilers, program analysis tools, and related utilities.[1] We have chosen LLVM because of its modular design, which reduces the effort to build the compilation tool chain for Patmos. LLVM, additionally, provides many technical advantages over other open-source compilers, such as GCC or Open64. Most notably, LLVM provides a high-level intermediate format, the LLVM bitcode, that can be used to perform aggressive optimisations on whole programs (often referred to as link-time optimisations). This capability is a prerequisite for compilers that aim at the optimisation of the worst-case execution time (WCET), as envisioned by the T-CREST project.

In this document, we present the principal components of the compiler tool chain and their interactions (Section 2). We discuss the current status of each component and give an overview of the changes and adaptations that were necessary to accommodate the special requirements of the Patmos processor and provide an outlook on the next steps planned. Section 3 contains a preliminary evaluation of the compilation tool chain, presenting first results of executing the integer benchmarks of the MiBench benchmark suite on the Patmos simulator. Measures we take for quality assurance are presented in Section 4. We include a tutorial on how to use the tools to compile a C program to Patmos machine code in Section 6. We conclude with an overview of the current state of the compiler and a description of how the requirements on the compiler are met by the current design in Section 5, and a conclusion in Section 8.

2 The Patmos Compiler

The project goal is to primarily support the C programming language, the efforts thus focus on providing a complete tool chain for this language in the context of an embedded system without an underlying operating system. However, the tool chain can easily be extended as requirements and needs evolve within the project. The principal components of the tool chain currently are: (1) the C frontend `Clang`,[2] (2) the LLVM optimisers, (3) an LLVM-based code generator and LLVM-based

[1] `http://www.llvm.org/`
[2] `http://clang.llvm.org/`

utilities for the Patmos processor, (4) the `gold` linker of the binutils project,[3] (5) the `newlib` C library,[4] and (6) the compiler support library `compiler-rt`.[5]

2.1 Tool Chain Overview

Figure 1 depicts a typical compilation flow of the Patmos tool chain. First, the C source code of a user-supplied application is translated to LLVM bitcode using the `clang` frontend. Next, the generated bitcode files are linked with the system libraries (e.g., the `newlib` C library) using the LLVM tool `llvm-ld`. This results in one combined bitcode file, which contains all user- and system code of the application. The *linked* application can then be optimised using the high-level optimisations of LLVM, which are readily available through `llvm-ld`. In the following step the linked and optimised bitcode is translated to machine code using the Patmos code generator and the LLVM tool `llc`. This results in a relocatable ELF binary, which already contains the entire machine code of the application and thus is, in principle, ready to be executed. The final step of the tool chain performs the final code and data layout of the application using the `gold` linker and an optional, user-supplied linker script. Note that no new code is added at this step since all libraries were already linked at the bitcode-level beforehand. This compilation flow was specifically designed to facilitate the development of future WCET-driven compilation techniques within the T-CREST project. Since all the application code is linked into a single bitcode file, the LLVM optimisers and the Patmos code generator have a complete view of all the application code needed to optimise the WCET.

The following section describes each of these components in more detail.

2.2 Principal Components

clang
: `clang` is the C frontend that translates source code files written in C to the LLVM intermediate representation (aka *bitcode*). The bitcode is the representation the LLVM analyses and optimisations work upon. `clang` also acts as a driver that invokes the tools involved with the options specific for targeting Patmos.

  Input: `.c` source code file
  Output: `.bc` bitcode object file

llvm-ld
: `llvm-ld` links bitcode files and static libraries of bitcode files together into a bitcode file containing the bitcode from the input files and the required symbol definitions from the libraries.

  Input: `.bc` bitcode object files [, `lib*.a` bitcode libraries]
  Output: `.bc` linked bitcode object file

llc
: `llc` constitutes the backend translating a bitcode file into either assembly or binary machine code for the Patmos ISA. The binary output format is relocatable ELF.

[3] `http://sourceware.org/binutils/`
[4] `http://sourceware.org/newlib/`
[5] `http://compiler-rt.llvm.org/`

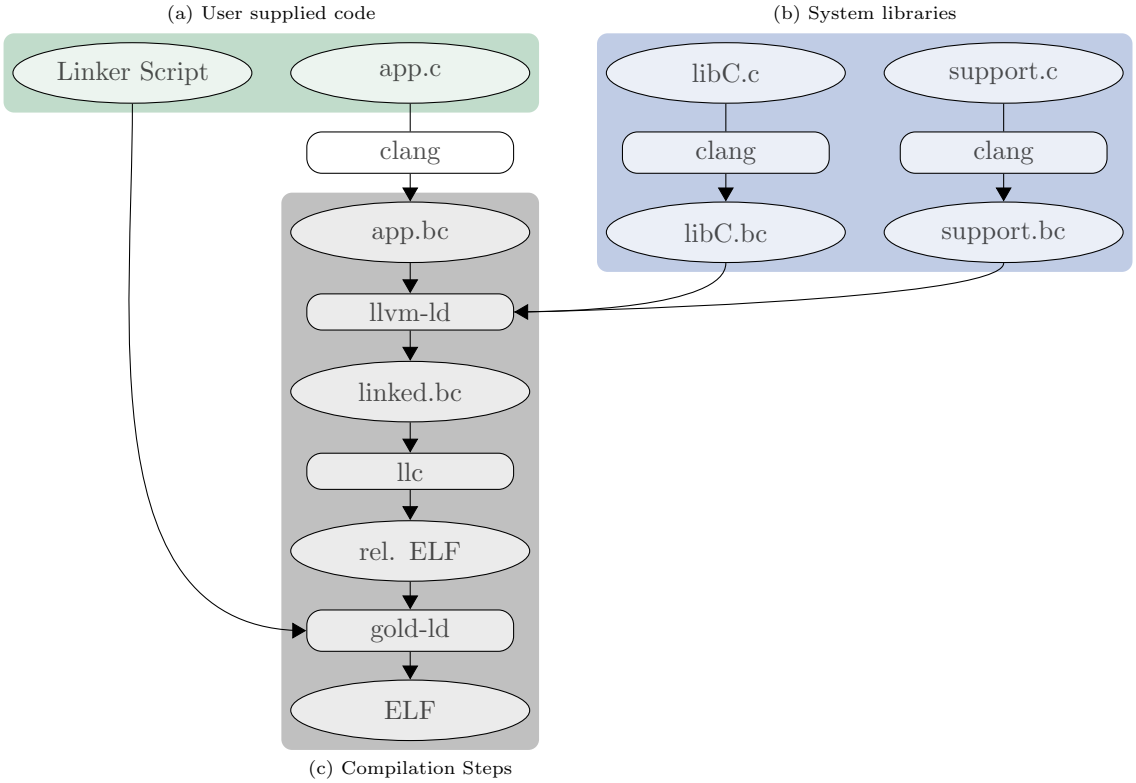


Figure 1: Compilation flow of the Patmos tool chain.

Input: `.bc` bitcode object file
Output: `.s` Patmos assembly
or `.o` Patmos relocatable ELF

gold

`gold` is an ELF linker. In our tool chain it is adopted to perform relocations.

Input: `.o` Patmos relocatable ELF
Output: `.o` Patmos final ELF

C Library

We adopted `newlib`, a standard ANSI C library following the C90 and C99 standards [4, 5], for Patmos. In the Patmos compilation flow the entire `newlib` C library is translated to LLVM bitcode and linked as such with the application program *before* code generation. This is an important precondition for WCET-driven compilation as envisioned in the T-CREST project, as it provides the compiler with a complete view of *all* code belonging to an application.

Support Library

The LLVM compiler requires a support library, which provides for instance emulation functions for floating-point operations. LLVM's `compiler-rt` support library has been adopted for Patmos and is, similar to the C library, compiled to LLVM bitcode and linked with the application program if needed.

2.3 Compiler Support for Patmos Features

In the following we give a summary of features specific to the Patmos processor, such as the stack and method cache [10], that need special support from the compiler and other parts of the tool chain.

2.3.1 Stack Cache

The compiler already supports an initial strategy to allocate data of a function's stack frame to the stack cache. At this point, no special support is required concerning the stack cache from other parts of the tool chain than the code generator.

Currently, all compiler-generated *spill* slots, which are guaranteed to be non-aliased, non-escaping stack objects, are allocated on the stack cache. All other stack objects are allocated to a shadow stack in the global memory. This is a simple, but for certain benchmarks already quite effective, allocation strategy, which allows to conduct early experiments to guide the hardware design of the stack cache. The instructions to manage the stack cache, i.e., `sres` to reserve stack space, `sens` to ensure the availability of data in the cache, and `sfree` to free stack space, are placed at all function entries and exits, as well as at all call-sites within functions that use the stack cache. Both, the data allocation and the instruction placement strategy are implemented in a way that allows easy adaptation for optimisations.

Future Extensions The current data allocation strategy does not perform an alias or escape analysis and thus cannot allocate objects on the stack cache where this information is uncertain. We plan to extend the allocation strategy accordingly in the scope of Task 5.2. Once information from the WCET analysis tool is available to the compiler, the allocation strategy can also be extended to selectively allocate and place data within the stack cache to reduce the WCET, during Task 5.3. This also applies to the placement of the stack management instructions.

2.3.2 Method Cache

The compiler and `gold` linker were extended to provide initial support for Patmos' method cache. The compiler currently allocates entire functions to the method cache, neither considering constraints on the code size nor effects on the task timing. Functions that do not fit into the method cache are split into subfunctions by the compiler. The compiler employs a greedy algorithm based on [12, 9] that partitions the control flow graph into single-entry regions which are then emitted as subfunctions.

Future Extensions In order to respect the size constraints of the method cache, several compiler parameters have to be optimised. For instance, the function in-lining optimisation will be revisited in Task 5.2 to account for the specific requirements of the method cache. In addition, splitting strategies will be developed that optimise the placement of code blocks within the cache to reduce the WCET during Task 5.3.

2.3.3 Typed Memory Operations

All components of the tool chain are capable of processing all typed memory operations. Bypassing the cache is already supported by using the `volatile` keyword in C. At the current stage, accessing the local memory is supported by manual programming of assembler (or inline assembly within C). This is done, for instance, to access I/O devices such as the UART within the `newlib` C library.

Future Extensions The manual or automatical allocation of data to the local memory is considered in Task 2.3 and 5.2. We plan to support user annotations to direct the allocation of data to the local memory. Automatic allocation strategies to exploit all kinds of typed memory operations are planned once the WCET analysis is able to provide corresponding information to the compiler, in Task 5.3.

2.3.4 Predicated Execution

All instructions of the Patmos ISA are predicated, i.e., an instruction only has an effect on the processor state if the specified 1-bit predicate register is set. Currently, predicated execution is mainly used for conditional branches and conditional move operations in the code generator, albeit all predicate operations of the Patmos ISA (compare, combine) are supported. The implementation of predicate spilling takes into account the fact that the predicate registers are contained in a special register. The code generator provides functions for branch- and predicate analysis and manipulation, which form the basis for if-conversion [1] and single-path code generation [8].

Future Extensions As planned for Task 5.2, the backend will be extended by a single-path code generator. We plan to adapt if-conversion to use information provided by WCET analysis instead of the commonly used heuristics and profiling data for region formation and scheduling in Task 5.3.

2.4 Additional Tool Chain Features

2.4.1 Assembler and Inline Assembly

The tool chain provides a full assembler and disassembler. Assembly files are compiled to Patmos relocatable ELF files with the `llvm-mc` tool. The `llvm-objdump` tool is used to disassemble Patmos ELF files for debugging purposes. The `clang` compiler also supports inline assembly similar to the GCC inline assembly syntax, such that hardware devices such as the UART device can be programmed at machine code level.

2.4.2 Support Libraries and Software Floating-Point Support

The adaption of the `newlib` library to the Patmos platform provides support for standard libraries such as `libc` and `libm` in the tool chain. Standard IO functions such `printf` can be used to communicate over the hardware UART provided by Patmos. The libraries additionally provide functions to read out a cycle counter and the time since boot-up.

The compiler supports floating point operations by generating calls to the software floating point implementation provided by the `compiler-rt` library. However, floating point support can be disabled for both the standard libraries and the compiler, so that the generated application does not contain any software floating point code, as this code could have a large negative impact on the WCET bound.

2.4.3 CMake Tool Chain

We also provide a tool chain configuration file for the build system `CMake`,[6] which automatically configures `CMake` for the Patmos tool chain. This allows for setting up a simple build system for Patmos applications with very little effort.

3 Preliminary Evaluation

In order to evaluate the current state of the compiler tool chain we compiled and executed a subset of the publicly available benchmark suite MiBench [3]. These benchmarks cover a representative set of tasks often encountered in embedded systems, e.g., in telecommunication and automotive domains. Since Patmos does currently not have native support for floating point operations, we limited the set of benchmarks to pure integer benchmarks only. Table 1 summarises the benchmarks used in the evaluation.

Each benchmark was first compiled – as illustrated by Figure 1 – to LLVM bitcode by the `clang` C frontend using optimisation level `-O3` and then linked and optimised using `llvm-ld`. Patmos machine code was generated using LLVM's `llc` tool using two configurations: (1) with stack cache

[6] `http://www.cmake.org/`

Benchmark	Domain	LOC	Description
bitcnts	automotive	938	Count the number of set bits in a word
qsort	automotive	47	Sort an input file of strings
dijkstra	network	350	Shortest paths search
patricia	network	621	Routing using Patricia trees
dbf	security	2524	Data decryption using blowfish
ebf	security	2524	Data encryption using blowfish
drijndael	security	2442	Data decryption using AES
erijndael	security	2442	Data encryption using AES
sha	security	269	Secure hashing
crc32	telecomm	291	Checksum calculation
rawcaudio	telecomm	741	ADPCM audio coding
rawdaudio	telecomm	741	ADPCM audio decoding
search-large	office	3054	Pratt-Boyer-Moore string search (large data set)
search-small	office	460	Pratt-Boyer-Moore string search (small data set)

Table 1: Short summary of benchmark programs used to evaluate the Patmos compiler tool chain.

support enabled and (2) with stack cache support disabled, i.e., all stack data is kept in the global main memory. The code layout is finalised by the `gold` linker using a default memory layout, where all code and data sections are placed into Patmos' global memory.

The benchmarks were executed using the Patmos simulator assuming a stack cache size of 1kB, a data cache size of 8kB, and a method cache size of 64kB. The stack cache is organised in word-sized blocks (4B), while the data and method caches are organised in 32-byte blocks. The 4-way set-associative data cache uses a least-recently-used (LRU) replacement policy and a write-through strategy with no-write allocation. The method cache likewise uses an LRU replacement policy. The cache size was chosen to accommodate the largest function in the benchmarks (`vf_printf`). Transferring a cache block (32B) to or from main memory is configured to take 40 cycles in the simulation. MiBench offers a small and a large data set for most benchmarks. To reduce the simulation overhead all benchmarks are run using the small data set, except for the `search` benchmark, where both the large and small data sets are used.

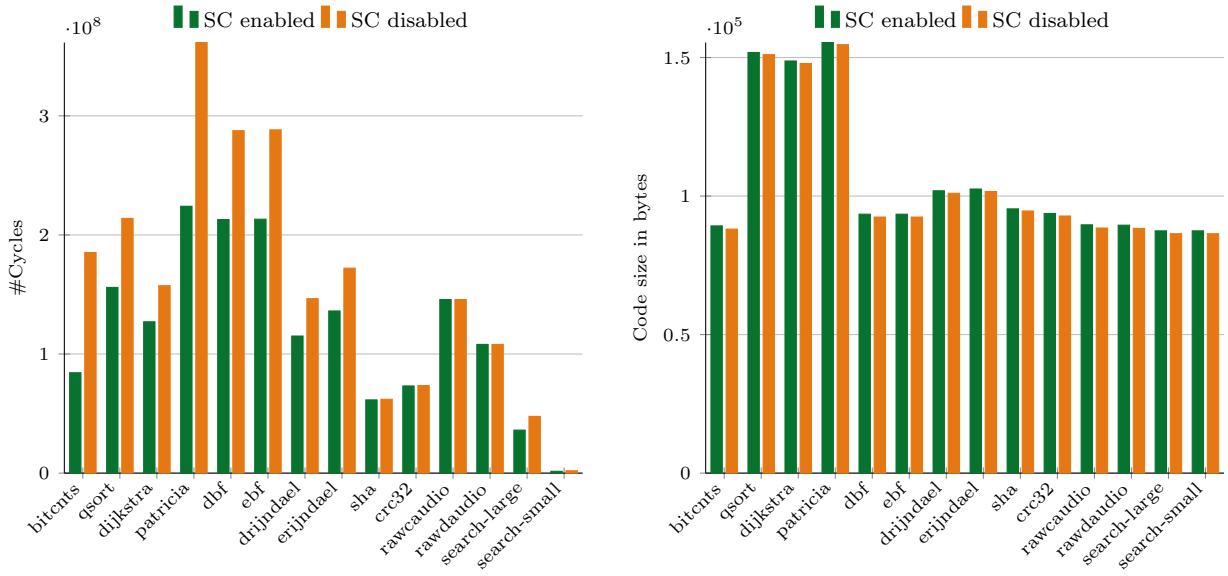
Since any specific Patmos optimisations will be implemented in Task 2.3, 5.2 and 5.3, we limit our discussion to some generic metrics, i.e., the total number of executed cycles, code size statistics, and statistics on the utilisation of the stack cache. These numbers are of course not final and will most likely change in future versions of the compiler. The main goal of this section is to demonstrate that the compiler tool chain, in its current state, is robust enough to be used for the upcoming developments within the T-CREST project.

3.1 Runtime and Code Size

In our first experiment we compare the total number of execution cycles for each benchmark with stack cache support enabled against a variant with the stack cache disabled. Enabling the stack cache allows us to (partially) allocate the stack frame of functions to the stack cache instead of the global memory. This reduces the number of accesses through the data cache, but at the same time increases the number of instructions, because the stack cache is explicitly managed using dedicated instructions by the compiler. As shown by Figure 2(a), the total number of execution cycles is reduced by enabling the stack cache for certain benchmarks. The gains are explained by (a) the additional cache space in the stack cache and (b) the handling of writes by the data cache. The additional cache space (1kB) in the stack cache reduces the number of loads from the data cache and thus the number of accesses to the slow main memory. Additional gains are due to the long latency of stores to the data cache, which uses a write-through strategy with no-write allocation.

Some benchmarks do not profit (yet) from the stack cache, most notably `crc32`, `rawcaudio`, and `rawdaudio`. These benchmarks are dominated by a single loop that does not contain any spill code or function calls (all system calls are in-lined). The simple stack cache allocation strategy is thus not able to find any data to allocate to the stack cache.

The runtime overhead of managing the stack cache is clearly out-weight by its gains. Figure 2(b) furthermore shows that the code size is only marginally increased by 1% in average and 1.3% in the worst-case. These numbers are expected to improve with more advanced stack cache allocation and code and data placement algorithms. Note that all benchmarks were compiled with full floating-point support enabled in the C library, which increases the overall code size.



(a) Number of execution cycles

(b) Code size in bytes

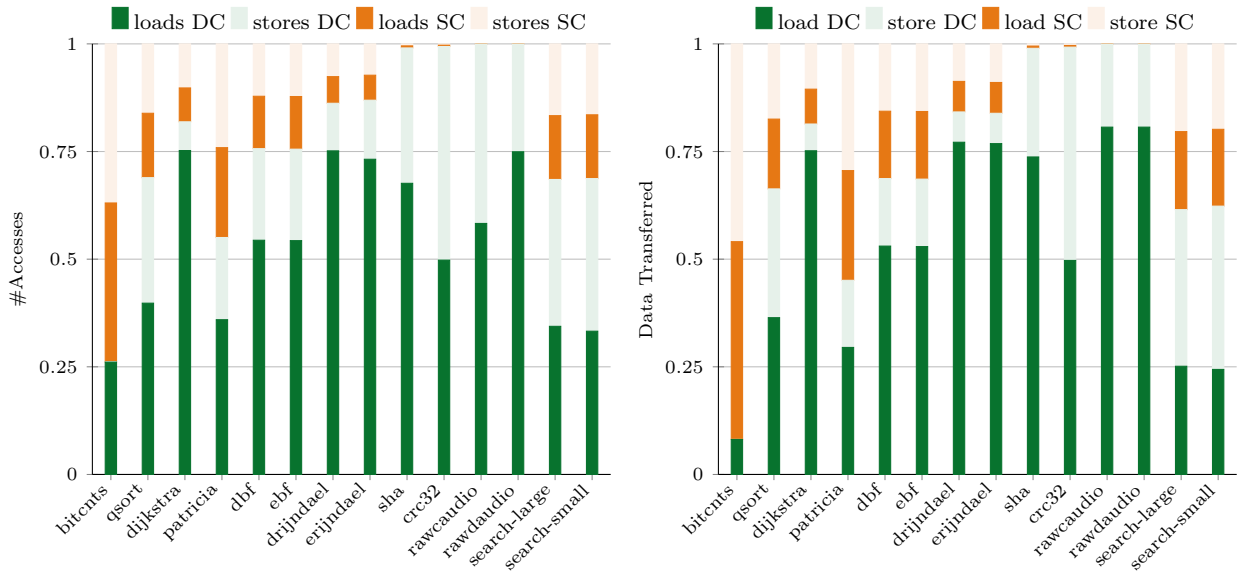
Figure 2: Comparison of each benchmark compiled with stack cache support enabled and disabled.

3.2 Stack Cache Utilisation

As shown before, utilising the stack cache can be quite profitable. We thus present some additional data characterising how the stack cache is currently used by the various benchmarks. Figure 3 highlights the number of dynamic memory accesses and the corresponding transfer volume[7] to and from the data and stack cache. The benchmarks that showed the best runtime improvements make heavy use of the stack cache. For instance, 75% of the memory accesses of the `bitcnts` go to the stack cache. With regard to data transfer volume, these numbers even increase. While roughly 45% of the accesses of the `patricia` benchmark target the stack cache, more then 50% of the transfer volume is serviced by the stack cache.

Benchmarks with little or no gain rarely make use of the stack cache, as shown by Figure 2. The main reason is that these benchmarks are dominated by a single loop without any spill code or function calls. This is confirmed by the numbers in Figure 4, which show a clear correlation between the amount of data dynamically allocated on the stack cache using `sres` instructions (Figure 4(a)) and the number of function calls executed in the benchmark (Figure 4(b)). Indeed, the current algorithm to allocate data on the stack cache only leverages compiler-generated spill slots, which are often linked to function calls (saving/restoring registers before and after calls). However, we expect that more powerful allocation algorithms will be able to overcome this limitation. For instance, the `rawcaudio` and `rawdaudio` benchmarks mostly operate on small buffers, which are potential candidates for stack cache allocation.

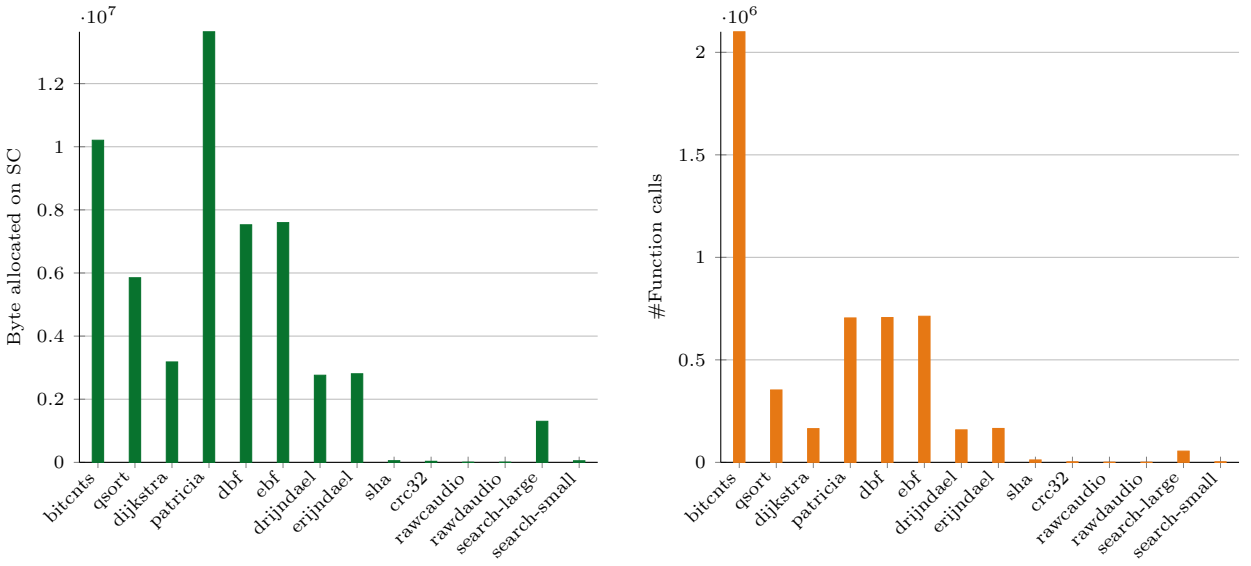
[7]Bytes read or written using load and store instructions.



(a) Normalised number of accesses

(b) Normalised transfer volume

Figure 3: Characteristics of data accesses to the data and stack cache for each benchmark.



(a) Data dynamically allocated on the stack cache.

(b) Number of dynamically executed function calls.

Figure 4: The stack cache usage correlates with the number of executed function calls.

4 Quality Assurance

The compiler plays a crucial role in the T-CREST project. It is therefore necessary to keep the compiler as stable as possible even while it is being developed. As shown in the previous section, the compiler tool chain is already able to handle larger, real-world benchmarks and, in combination with the Patmos simulator, already provides first insights and feedback to be used to guide the upcoming developments within the T-CREST project. In this section we present measures that have been taken to ensure the stability of the tool chain while new features are being added.

4.1 Continuous Testing

We use dedicated servers at the TUV and at DTU to continuously test the latest modifications to the compiler and related tools, such as the linker and simulator. These servers build and run several configurations of the tool chain. The tool chain is then used in turn to build and simulate several large benchmarks such as the MiBench benchmarks and the Mediabench benchmarks. The output of the benchmarks is compared against reference outputs. Simulation statistics and any problems that occur during compilation are recorded and thus provide continuous feedback on the current state of the tool chain development. For instance, the data discussed in the preceding sections is automatically gathered on each test run and sent out in daily reports.

4.2 Buildbot Infrastructure

Buildbot is a continuous integration system designed to automate the build/test cycle.[8] We have set up a Buildbot system on a server at TUV. It periodically polls the set of source-code repositories for changes and rebuilds those that contain changes. Additionally to these incremental builds, the whole tool chain is built from scratch daily. By building the tool chain on different architectures, the Buildbot infrastructure allows to pinpoint build problems quickly. It is very easy to extend the system by machines of project partners to ensure that they are able to build and work with the compiler tool chain. Conveniently, Buildbot has a variety of status-reporting tools, including a web-frontend, mail-notifications and an IRC-bot, to obtain information about the latest builds and to access the generated binaries. The web-interface of our Buildbot system is accessible at `http://buildbot.vmars.tuwien.ac.at/patmos/` Figure 5 shows a screenshot of the Buildbot web-interface.

5 Current State and Outlook

> "**D 5.2: Initial compiler version (month 12)** LLVM adaption for the Patmos processor, providing support of single-path code generation."

We summarise the current state of the compiler work package with respect to the fulfilment of the deliverable's requirements as follows:

[8] `http://www.buildbot.net`

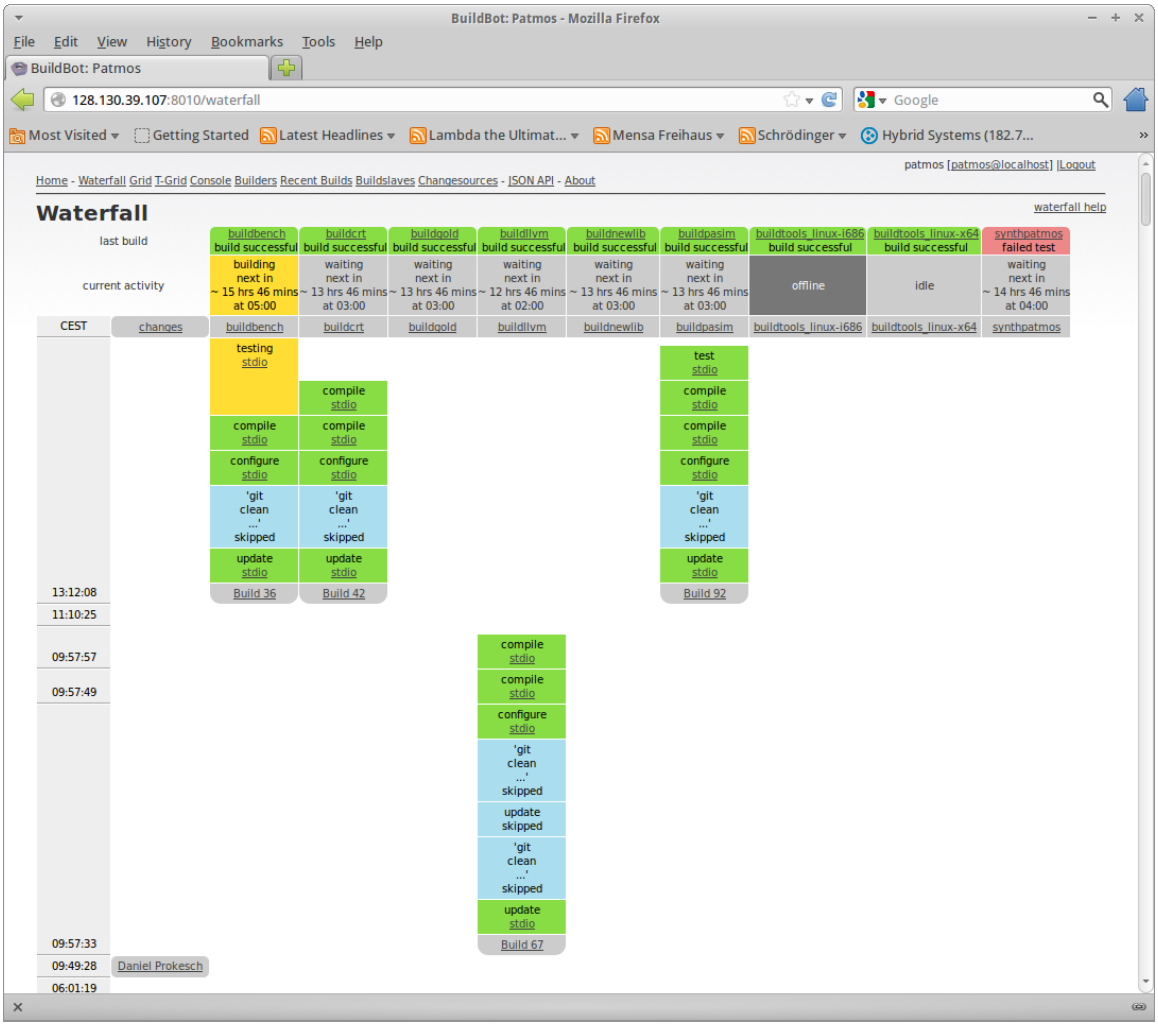


Figure 5: The Buildbot web-interface displays information about the latest builds.

- **Stable C compiler tool chain**
  The compiler tool-chain components developed and adapted for the Patmos processor are in a state in which it is possible to compile established benchmark applications, and to execute them on the simulator provided in D2.1.
- **Library support**
  We have ported an implementation (newlib) of the standard libraries libc and libm. As Patmos does not provide an FPU, soft-float libraries have been adapted to allow compilation of C code that uses floating-point arithmetic.
- **Validation and testing**
  To provide Quality Assurance, a building and testing infrastructure has been set up that regularly builds the compiler tool chain upon changes and verifies its functional correctness on the basis of a set of benchmark suites.
- **Compiler features**
  The compiler provides basic support for distinguishing features of Patmos, particularly the stack cache, the method cache, and the dual-issue pipeline. The code-generation backend, which supports the complete fully-predicated Patmos ISA, is able to emit both text assembly and object code. An assembler and a disassembler are part of the tool chain.
- **Single-path support**
  The code-generation backend supports predication of all instructions. Functions for branch and predicate analysis and manipulation are implemented, which provide the basis for single-path code generation.
- **WCET-analysis support**
  The tool chain exports symbol information to the generated ELF binary, including label addresses, function sizes, and relocation types. All branches except those resulting from jump tables are translated into direct branches, which are simple to analyse. Exporting additional control flow information to the WCET analysis (Task 6.1) is not yet supported, instead it is planned to add this support as part of the WCET analysis integration task (Task 5.3).
- **Optimisations**
  At this point, the generic optimisations provided by LLVM are available, which can be selectively enabled or disabled by command-line options.

The current tool chain framework provides the foundation for the extensions that are scheduled for the upcoming tasks of the T-CREST project:

- **Scheduling for both pipelines**
  To benefit from the VLIW architecture of Patmos, the instruction scheduler will be extended to utilise both instruction pipelines and to reorder instructions to minimise stalling during memory accesses. (Task 5.2)
- **Single-path code generator**
  One aspect of time-predictability is stability. A single-path code generator will be developed to achieve stable task execution times. (Task 5.2)
- **Typed memory access and optimisations targeting the memory hierarchy**
  The split-cache design of Patmos demands for strategies to allocate data and schedule access to the different memories. For example, WCET oriented allocation strategies for the stack cache and a WCET oriented function splitting algorithm for the method cache will be developed.

These optimisations targeting predictability will be integrated into the compiler. (Task 5.2, 2.3)

- **WCET-analysis integration**
  The backend will be extended by a module that exports the relevant control-flow information on machine code level (e.g., targets of indirect branches) for the WCET analysis in the aiT specific **.ais** format. Furthermore, flow information annotated on source-code level will be co-transformed to machine-level code. The integration of the timing-analysis tool will provide the basis for WCET-guided optimisations. (Task 5.3)
- **WCET-aware optimisations**
  The optimisations provided by LLVM will be extended by WCET oriented optimisations, which take the features and properties of the Patmos architecture into account and may use feedback from the WCET analysis for guidance. (Task 5.3)

6 Installation and Usage of the Compiler

This section gives a guide for building and installing the Patmos tool chain and describes how to use it to compile applications for Patmos.

6.1 Prerequisites

The Patmos compiler tool chain is developed for the Linux platform. In order to build all tools, the following programs and libraries are required:

cmake, make
: The CMake (at least version 2.8) and `make` build systems are required to build the various components of the tool chain, such as the `clang` compiler or the `compiler-rt` system library.

gcc, g++
: A C and C++ compiler such as GCC is required to build `clang`, `llvm` and `gold`. It is also possible to use a separate `clang` installation as an alternative to GCC. Compiling with `clang` will result in shorter compile times, however, if the compiled application should be debugged with `gdb`, it is recommended to use `gcc` and `g++`.

flex, bison, texinfo
: These tools are required to build tools such as gold successfully.

libelf
: The tools in the Patmos tool chain require the development headers of `libelf`, as this library is used to read and write ELF files.

boost
: The simulator requires the program-options library from the `boost` C++ library (at least version 1.46).

git
: The version control system `git` is required to download the latest development versions of the tools.

On current Debian based Linux distributions, the following command can be used to install all the required packages:

```
sudo apt-get install git cmake make g++ texinfo flex bison \
  libelf-dev graphviz libboost-dev libboost-program-options-dev
```

6.2 Retrieving and Building the Source Code

We provide a build script that retrieves, configures and builds all the required packages automatically (cf. Section 6.2.1). Alternatively, the various tool chain source repositories can be acquired individually from the `t-crest` organisation at `github.com`.[9], as described in Section 6.2.2.

We created a tag called `release_m12` in all tool chain related source repositories mentioned in this report that marks the current stable release version of the various tools at the time of writing. To build the released compiler version instead of the latest development version, execute the following command in all of the source repositories after they have been checked out, and then rerun the build commands.

```
git checkout release_m12
```

Alternatively, `github.com` provides a link in all of the source repositories to download the source code for that tag as zip files.

For the following build instructions we use directory structure presented in Table 2. We use `~/tcrest/` as root directory and `~/tcrest/local/` as installation directory in this document. However, the root directory and the installation directory can be chosen arbitrarily.

6.2.1 Building the Tool Chain Using the Build Script

After installing the prerequisites, download or check out the build script from the `patmos-misc` repository on `github.com`:

```
git clone https://github.com/t-crest/patmos-misc.git misc
```

To configure the build script, place a configuration file called `build.cfg` into the same directory as the `build.sh` script. In this configuration file, set the `ROOT_DIR` variable to the directory which should contain all the repository checkouts and build directories (e.g., `ROOT_DIR=~/tcrest/`).

[9] `http://github.com/t-crest/`

Directory	Description
~/tcrest/	Base directory (ROOT_DIR)
~/tcrest/patmos	Processor source repository
~/tcrest/patmos/simulator	Simulator sources
~/tcrest/patmos/simulator/build/	Simulator build directory
~/tcrest/compiler-rt/	System library source repository
~/tcrest/compiler-rt/build/	System library build directory
~/tcrest/gold/	ELF linker and binutils sources
~/tcrest/gold/build/	Linker and binutils build directory
~/tcrest/llvm/	LLVM source repository
~/tcrest/llvm/tools/clang	Clang source repository. Must always be a tools/clang subdirectory of LLVM
~/tcrest/llvm/build/	Build directory for LLVM and clang
~/tcrest/local/	Install base directory (INSTALL_DIR)
~/tcrest/local/bin/	Install directory for tool chain binaries. All binaries in this directory are prefixed with patmos- by the build script
~/tcrest/local/lib/	Install directory for tool chain libraries
~/tcrest/local/patmos-unknown-elf/	Install directory for Patmos libraries and headers
~/tcrest/misc/	Support files (e.g., build script, config files)
~/tcrest/newlib/	Sources of standard C library for Patmos
~/tcrest/newlib/build/	Standard library build directory

Table 2: Default directory structure of the tool chain

```
cd misc
cp build.cfg.dist build.cfg
# edit build.cfg, set ROOT_DIR and INSTALL_DIR
./build.sh
```

Executing `build.sh` checks out any missing repository, configures and then builds all the required tools and libraries of the tool chain. By default, the directory structure created by the build script follows the directory structure presented in Table 2. All necessary compiler tools are installed to `INSTALL_DIR/bin`, while Patmos specific libraries such as `newlib` and `compiler-rt` are installed under `INSTALL_DIR/patmos-unknown-elf/`. All executables are prefixed with `patmos-` to differentiate them from non-Patmos related installations, e.g., `clang` will be installed as `patmos-clang`.

6.2.2 Building the Tool Chain Manually

To build all tools and libraries manually, check out and build the following repositories as described below.

The Binary Linker `gold`

The following commands build the `gold` linker for Patmos ELF files.

```
git clone https://github.com/t-crest/patmos-gold.git gold
mkdir gold/build
cd gold/build
../configure --program-prefix=patmos- --enable-gold=yes \
    --enable-ld=no --enable-plugins --prefix=~/tcrest/local
make all-gold install-gold
```

The Compiler Framework `llvm` and the C Compiler Tool `clang`

The `patmos-clang` repository must be checked out inside the `patmos-llvm` repository checkout as `tools/clang` directory (creating a symlink instead is not supported). `Clang` is then built automatically when `llvm` is built.

```
git clone https://github.com/t-crest/patmos-llvm.git llvm
git clone https://github.com/t-crest/patmos-clang.git \
                                          llvm/tools/clang
mkdir llvm/build
cd llvm/build
cmake -DLLVM_TARGETS_TO_BUILD=Patmos \
      -DCMAKE_BUILD_TYPE=Debug \
      -DCMAKE_INSTALL_PREFIX=~/tcrest/local ..
make all install
```

The C Library `newlib`

Newlib uses the `llvm-ar` and `llvm-ranlib` tools provided by LLVM to build bitcode archives, while `clang` is used to link files.

```
git clone https://github.com/t-crest/patmos-newlib.git newlib
mkdir newlib/build
cd newlib/build
../configure --target=patmos-unknown-elf \
    AR_FOR_TARGET=~/tcrest/local/bin/llvm-ar \
    RANLIB_FOR_TARGET=~/tcrest/local/bin/llvm-ranlib \
    LD_FOR_TARGET=~/tcrest/local/bin/clang \
    CC_FOR_TARGET=~/tcrest/local/bin/clang \
    CFLAGS_FOR_TARGET="-target patmos-unknown-elf -O2" \
    --prefix=~/tcrest/local
make all install
```

Alternatively, the `ar` and `ranlib` tools provided by `patmos-gold` can be used to build `newlib`, provided that LLVM is configured to build the binutils link time optimisation (LTO) plugin that provides bitcode file support for the binutils tools.

The Support Library `compiler-rt`

The following commands build and install the runtime libraries for Patmos.

```
git clone https://github.com/t-crest/patmos-compiler-rt.git \
                                                    compiler-rt
mkdir compiler-rt/build
cd compiler-rt/build
cmake -DCMAKE_TOOLCHAIN_FILE=\
        ../cmake/patmos-clang-toolchain.cmake \
      -DCMAKE_PROGRAM_PATH=~/tcrest/local/bin \
      -DCMAKE_INSTALL_PREFIX=~/tcrest/local ..
make all install
```

The tool chain file `patmos-clang-toolchain.cmake` sets up the build environment with the tools of the Patmos tool chain that are found in the given program path. The compiler-rt libraries are installed along with the newlib libraries into the default library lookup path of the Patmos compiler.

The Patmos Simulator `pasim`

The simulator for Patmos was part of Deliverable 2.1 and is part of the Patmos processor repository. Compiling the simulator requires an installation of the development headers of libboost and libelf. The following commands build and install the simulator.

```
git clone https://github.com/t-crest/patmos.git
mkdir patmos/simulator/build
cd patmos/simulator/build
cmake -DCMAKE_INSTALL_PREFIX=~/tcrest/local ..
```

For more information on building and using the tools, please refer to the `README.patmos` files that are contained in the source repositories.

6.3 Running the Compiler

The compiler is executed by running the compiler driver tool `patmos-clang`. This invokes the necessary tools to compile, assemble and link an application.

In order to build an application for Patmos, the target platform triple, the output file name and the source files (either C code or bitcode files) need to be provided to `patmos-clang`.

```
patmos-clang -target patmos-unknown-elf -o hello main.c hello.c
```

This will compile the source files to bitcode files, link in `newlib` and the system libraries, and execute `patmos-gold` to create the final executable application. If the option `-v` is passed to `patmos-clang`, the tools print out all commands that are executed in the background, as well as additional information about search paths and linked libraries.

After compilation, the application is executed in the simulator by using

```
pasim hello
```

6.3.1 Driver Options

The `patmos-clang` driver can also be used to generate bitcode files, to link bitcode files, or to emit assembler code. The driver supports the following modes of operation (for brevity, we omit the `-target patmos-unknown-elf` argument):

`patmos-clang -c <inputs>`
- Input: `.c` C source file
- Output: `.o` or `.bc` bitcode files
- Actions: compile each input file to a bitcode file

`patmos-clang -S <inputs>`
- Input: `.c` C source file
- Output: `.s` or `.ll` human readable bitcode files
- Actions: compile each input file to a human readable bitcode file

`patmos-clang -emit-llvm <inputs>`
- Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
- Output: bitcode file
- Actions: compile to bitcode, link all input files, link with standard libraries and start code

Option	Description
`-mfloat-abi=none`	Do not use software floating point libraries when linking
`-nostdlib`	Do not use standard libraries such as `libc` when linking
`-nolibc`	Do not use `libc` when linking
`-nodefaultlibs`	Do not use platform system libraries when linking
`-nostartfiles`	Do not use the `crt0` start file when linking
`-nolibsyms`	Do not use symbol definition files for runtime libraries when linking. Those files prevent the linker from removing any functions for which calls might be generated by the compiler backend, such as software division or `memcpy`
`-fpatmos-link-object`	Link as object, i.e., do not link in any libraries or start code

Table 3: Options for `patmos-clang` that control the default behaviour of the linker

`patmos-clang -fpatmos-emit-obj -c <inputs>`
Input: `.c` C source file
Output: `.o` Patmos relocatable ELF
Actions: compile each input file to a Patmos relocatable ELF file

`patmos-clang -fpatmos-emit-obj -S <inputs>`
Input: `.c` C source file
Output: `.s` Patmos assembly file
Actions: compile each input file to a Patmos assembly file

`patmos-clang -fpatmos-emit-obj <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: `.o` Patmos relocatable ELF
Actions: compile to bitcode, link all input files, link with standard libraries and start code, compile to relocatable ELF

`patmos-clang -fpatmos-emit-asm <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: `.o` Patmos assembly file
Actions: compile to bitcode, link all input files, link with standard libraries and start code, compile to Patmos assembly

`patmos-clang -o <output> <inputs>`
Input: `.c` C source file, `.bc` bitcode object file, `.a` bitcode files archive
Output: Patmos executable ELF
Actions: compile to bitcode, link with standard libraries and start code, compile to relocatable ELF, create Patmos executable ELF

The compiler accepts standard options such as `-I`, `-L` and `-l` to define additional lookup paths for header files and libraries and to link with (static) libraries. The behaviour of the linker can be controlled with additional options for `patmos-clang` as shown in Table 3.

6.3.2 Libraries

The Patmos tool chain supports static libraries. Libraries are archives that contain bitcode files. The archives are created by using either the `ar` tool provided by the host system or by using `patmos-ar` from the `patmos-gold` binutils. The tool `patmos-llvm-nm` can be used to inspect the content of the archive.

```
ar q libtest.a *.bc
# show the contents of libtest.a
patmos-llvm-nm libtest.a
# compile and link with the created library
patmos-clang -target patmos-unknown-elf -o app main.c -ltest
```

7 Requirements

We now list the requirements from Deliverable D 1.1 that target the compiler work package (WP5) and explain how they are met by the current initial version of the tool chain or how they will be addressed in future work. NON-CORE and FAR requirements are not listed here.

7.1 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).

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).

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. Optimisations utilising the features of the Patmos processor to improve the execution time are scheduled for Task 5.2. 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.

Extending the `clang` compiler to support memory type annotations in the source code is considered in Task 2.3. The annotations will be passed down to the compiler backend to guide the code generation so that data will be placed in the scratchpad memory instead of in the data cache.

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

Enhancing the LLVM scheduler for the Patmos instruction set architecture (ISA) is considered in Task 5.2. Instructions will be reordered to make optimal 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 as `volatile` 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 information about the LLVM compiler can be found in the LLVM user guide.[10]

7.2 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. Additional optimisations that make best use of the different memory types and support for annotations to select the memory type for memory accesses are considered in Task 2.3 and 5.2.

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

Extending the `clang` compiler to support memory type annotations in the source code is considered in Task 2.3. The annotations will be passed down to the compiler backend to guide the code generation so that memory accesses use the correct load and store instructions.

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.

All Patmos features are supported at the assembler level by the compiler. The compiler uses special optimisations to generate code that uses the method cache and the stack cache. Further optimisations that use the features provided by Patmos to reduce the WCET will be implemented in Task 5.3 and Task 2.3.

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

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

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 (esp. from Task 5.2) 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 `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.

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

Extending the `clang` compiler to export flow information to the WCET analysis tool is considered in Task 5.3. The compiler will emit both annotations provided by the user as well as internal information such as targets of indirect jumps for jump tables.

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

Extending the `clang` compiler to export flow information to the WCET analysis tool is considered in Task 5.3. The compiler will emit both annotations provided by the user as well as internal information such as targets of indirect jumps for jump tables.

8 Conclusion

The actual version of the Patmos compiler tool chain satisfies all essential requirements that have to be met at the current project stage. It is stable enough to compile large real-world benchmarks correctly for Patmos, as demonstrated by a preliminary evaluation of the MiBench benchmark suite. Continuous automatic testing is employed and proved useful to avoid any regressions during development. The compiler framework has initial support for special features of the Patmos architecture such as the stack cache and the method cache, and provides a good platform for the implementation of single-path code generation and WCET oriented code optimisations in the upcoming Task 5.2 and Task 5.3. The aiT tool will be integrated into the tool chain in order to provide flow facts to the WCET analysis and to perform WCET driven optimisations in Task 5.3. Optimisations for typed memory accesses and for the memory hierarchy of Patmos are in the scope of Task 5.2 and Task 2.3.

Appendix

A Building a Hello World Application

In this section we show how to compile and simulate a simple Hello World application with the Patmos tool chain, and present the output of the various tools.

The program we use in this example is

```
$ cat hello.c


#include <stdio.h>

int main(int argc, char** argv) {

    int i = 20;

    if (argc > 0) {
        i = i / argc;
    }

    printf("Hello World: %d\n", i);

    return 0;
}
```

The following command compiles the file `hello.c` into bitcode, using a human readable output format:

```
$ patmos-clang -target patmos-unknown-elf -o hello.ll -S hello.c
$ cat hello.ll


; ModuleID = 'hello.c'
target datalayout =
    "E-S32-p:32:32:32-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f64:32:64-n32"
target triple = "patmos-unknown-elf"

@.str = private unnamed_addr constant [17 x i8]
                            c"Hello World: %d\0A\00", align 1

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 4
  store i32 20, i32* %i, align 4
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %1 = load i32* %i, align 4
  %2 = load i32* %argc.addr, align 4
  %div = sdiv i32 %1, %2
  store i32 %div, i32* %i, align 4
  br label %if.end
```

```
if.end:                                           ; preds = %if.then, %entry
  %3 = load i32* %i, align 4
  %call = call i32 (i8*, ...)*
    @printf(i8* getelementptr inbounds ([17 x i8]* @.str, i32 0, i32 0), i32 %3)
  ret i32 0
}

declare i32 @printf(i8*, ...)
```

The following command compiles the file `hello.c` into Patmos assembly code:

```
$ patmos-clang -target patmos-unknown-elf -o hello.s \
                       -fpatmos-emit-asm -S hello.c
$ cat hello.s

 .file "/tmp/hello-bfRl1Q.bc"
 .text
 .globl main
 .type main,@function
 .fstart main, .Ltmp0-main, 4      # @main
main:
# BB#0:                                # %entry
        sres  3                       # encoding: [0x03,0x00,0x00,0x03]
        sub   $r31 = $r31, 32         # encoding: [0x00,0x7f,0xf0,0x20]
        mfs   $r9 = $sz               # encoding: [0x02,0x52,0x00,0x30]
        sws   [2] = $r9               # encoding: [0x02,0xc0,0x04,0x82]
        mfs   $r9 = $so               # encoding: [0x02,0x52,0x00,0x35]
        sws   [1] = $r9               # encoding: [0x02,0xc0,0x04,0x81]
        mfs   $r9 = $sb               # encoding: [0x02,0x52,0x00,0x34]
        sws   [0] = $r9               # encoding: [0x02,0xc0,0x04,0x80]
        swc   [$r31 + 2] = $r0        # encoding: [0x02,0xc5,0xf0,0x02]
        swc   [$r31 + 3] = $r3        # encoding: [0x02,0xc5,0xf1,0x83]
        swc   [$r31 + 4] = $r4        # encoding: [0x02,0xc5,0xf2,0x04]
        li    $r1 = 20                # encoding: [0x00,0x02,0x00,0x14]
        swc   [$r31 + 5] = $r1        # encoding: [0x02,0xc5,0xf0,0x85]
        li    $r1 = 1                 # encoding: [0x00,0x02,0x00,0x01]
        lwc   $r2 = [$r31 + 3]        # encoding: [0x02,0x85,0xf1,0x03]
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
        cmplt $p1 = $r2, $r1          # encoding: [0x02,0x02,0x20,0xb2]
 ( $p1) bc    .LBB0_2@FREL            # encoding: [0x0e,0b01AAAAAA,A,A]
        #   fixup A - offset: 0, value: .LBB0_2@FREL, kind: FK_Patmos_frel_22
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
# BB#1:                                           # %if.then
        lwc   $r3 = [$r31 + 5]        # encoding: [0x02,0x87,0xf1,0x05]
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
        lwc   $r4 = [$r31 + 3]        # encoding: [0x02,0x89,0xf1,0x03]
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
        bs    __divsi3                # encoding: [0x06,0b00AAAAAA,A,A]
        #   fixup A - offset: 0, value: __divsi3, kind: FK_Patmos_22
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
        nop   0                       # encoding: [0x02,0x40,0x00,0x00]
        sens  3                       # encoding: [0x03,0x40,0x00,0x03]
```

```
        swc   [$r31 + 5] = $r1      # encoding: [0x02,0xc5,0xf0,0x85]
.LBB0_2:                                        # %if.end
        lwc   $r1 = [$r31 + 5]      # encoding: [0x02,0x83,0xf1,0x05]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        swc   [$r31 + 1] = $r1      # encoding: [0x02,0xc5,0xf0,0x81]
        li    $r1 = .L.str          # encoding: [0x87,0xc2,0x00,0x00,A,A,A,A]
        #   fixup A - offset: 0, value: .L.str, kind: FK_Patmos_32
        swc   [$r31] = $r1          # encoding: [0x02,0xc5,0xf0,0x80]
        bs    printf                # encoding: [0x06,0b00AAAAAA,A,A]
        #   fixup A - offset: 0, value: printf, kind: FK_Patmos_22
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        sens  3                     # encoding: [0x03,0x40,0x00,0x03]
        mov   $r1 = $r0             # encoding: [0x00,0x02,0x00,0x00]
        lws   $r9 = [2]             # encoding: [0x02,0x92,0x00,0x02]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        mts   $sz = $r9             # encoding: [0x02,0x40,0x90,0x20]
        lws   $r9 = [1]             # encoding: [0x02,0x92,0x00,0x01]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        mts   $so = $r9             # encoding: [0x02,0x40,0x90,0x25]
        lws   $r9 = [0]             # encoding: [0x02,0x92,0x00,0x00]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        mts   $sb = $r9             # encoding: [0x02,0x40,0x90,0x24]
        sfree 3                     # encoding: [0x03,0x80,0x00,0x03]
        add   $r31 = $r31, 32       # encoding: [0x00,0x3f,0xf0,0x20]
        ret                         # encoding: [0x07,0x80,0x00,0x00]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
        nop   0                     # encoding: [0x02,0x40,0x00,0x00]
.Ltmp0:
.Ltmp1:
 .size main, .Ltmp1-main

 .type .L.str,@object                # @.str
 .section .rodata.str1.1,"aMS",@progbits,1
.L.str:
 .asciz  "Hello World: %d\n"
 .size .L.str, 17
```

We now show how to link the previously generated bitcode file with the standard libraries and start code and compile it to a Patmos ELF executable. The `-target` option can be omitted here since the target information is already stored in the bitcode file. The executable is then simulated using the `pasim` simulator.

```
$ patmos-clang -o hello hello.ll
$ pasim -M fifo -m 64k hello

Hello World: 20

Cyc : 3422
 PRR: 00000001  BASE: 0000efb8   PC : 0000efcc   <_exit:.LBB28_1>
 r0 : 00000000   r1 : 00000000   r2 : 00016890   r3 : 00000000
```

```
 r4 : 00000000   r5 : 00000408   r6 : 00000360   r7 : 00000400
 r8 : 00000409   r9 : 00001258   r10: 00000058   r11: 00000004
 r12: 00014d48   r13: 03ffff90   r14: 00000000   r15: 00000000
 r16: 00000000   r17: 00000000   r18: 00000000   r19: 00000000
 r20: 00000000   r21: 00000000   r22: 00000000   r23: 00016050
 r24: 00000001   r25: 00016198   r26: 00000000   r27: 00000000
 r28: 00000000   r29: 03fffe80   r30: 00000000   r31: 04000000

 s0 : 00000001   s1 : 00000000   s2 : 00000000   s3 : 00000000
 s4 : 00000000   s5 : 00000298   s6 : 03000000   s7 : 00000d5d
 s8 : 00000000   s9 : 00000000   s10: 00000000   s11: 00000000
 s12: 00000000   s13: 00000000   s14: 00000000   s15: 00000000


Instruction Statistics:
       instruction:   #fetched   #retired #discarded
              addi:         358         348          10
              subi:          78          78           0
             rsubi:           9           9           0
               sli:           1           1           0
               sri:           7           7           0
              srai:           2           2           0
               ori:          12          12           0
              andi:          36          36           0
              addl:          80          80           0
              subl:           0           0           0
             rsubl:           1           1           0
               sll:           0           0           0
               srl:           0           0           0
              sral:           0           0           0
               orl:           1           1           0
              andl:          12          12           0
               rll:           0           0           0
               rrl:           0           0           0
              xorl:           6           6           0
              norl:           0           0           0
            shaddl:           0           0           0
           shadd2l:           0           0           0
               add:          57          57           0
               sub:          49          49           0
              rsub:           0           0           0
                sl:           1           1           0
                sr:           0           0           0
               sra:           0           0           0
                or:           1           1           0
               and:           3           3           0
                rl:           0           0           0
                rr:           0           0           0
               xor:           0           0           0
               nor:           0           0           0
             shadd:           0           0           0
            shadd2:           0           0           0
```

```
   sext8:          0          0          0
  sext16:          3          3          0
  zext16:          4          4          0
     abs:          0          0          0
     mul:          3          3          0
    mulu:          2          2          0
   cmpeq:        129        129          0
  cmpneq:         82         82          0
   cmplt:         30         30          0
   cmple:          2          2          0
  cmpult:         38         38          0
  cmpule:         14         14          0
   btest:         29         29          0
     por:          0          0          0
    pand:          0          0          0
    pxor:          0          0          0
    pnor:         17         17          0
     nop:       1138       1136          0
    wait:          0          0          0
     mts:         43         43          0
     mfs:         49         49          0
     lws:        122        122          0
     lwl:          0          0          0
     lwc:        111        111          0
     lwm:          0          0          0
     lhs:          0          0          0
     lhl:          0          0          0
     lhc:          7          7          0
     lhm:          0          0          0
     lbs:          0          0          0
     lbl:         16         16          0
     lbc:          1          1          0
     lbm:          0          0          0
    lhus:          0          0          0
    lhul:          0          0          0
    lhuc:         12         12          0
    lhum:          0          0          0
    lbus:          0          0          0
    lbul:          0          0          0
    lbuc:         64         64          0
    lbum:          0          0          0
    dlwc:          0          0          0
    dlwm:          0          0          0
    dlhc:          0          0          0
    dlhm:          0          0          0
    dlbc:          0          0          0
    dlbm:          0          0          0
   dlhuc:          0          0          0
   dlhum:          0          0          0
   dlbuc:          0          0          0
   dlbum:          0          0          0
     sws:        139        139          0
     swl:          0          0          0
```

```
               swc:        115         115           0
               swm:          0           0           0
               shs:          0           0           0
               shl:          0           0           0
               shc:         22          22           0
               shm:          0           0           0
               sbs:          0           0           0
               sbl:         16          16           0
               sbc:         19          19           0
               sbm:          0           0           0
              sres:         19          19           0
              sens:         20          20           0
             sfree:         18          18           0
                bs:         18          18           0
                bc:        362         192         170
                 b:          0           0           0
               bsr:          5           5           0
               bcr:          1           1           0
                br:          0           0           0
               ret:         22          21           0
               bne:          0           0           0
               all:       3406        3223         180
           bubbles:          -        3441           -

Stall Cycles:
   DR: 16
   EX: 0
   MW: 0


Method Cache Statistics:
                          total        max.
   Blocks Transferred:     1465         951
   Bytes Transferred :    46708       30432
   Cache Hits        :       29
   Cache Misses      :       15
   Miss Stall Cycles :        0

       Method:      #hits     #misses
   0x00001078:          1           0    <_start>
   0x000010c4:          1           1    <main>
   0x00001258:          1           1    <exit>
   0x0000164c:          2           1    <memmove>
   0x00001988:          1           1    <printf>
   0x00001a0c:          4           1    <_vfprintf_r>
   0x000090f0:          0           1    <__sinit>
   0x0000962c:          7           1    <_cleanup_r>
   0x000098c4:          5           1    <__sprint_r>
   0x0000af1c:          1           1    <__swsetup_r>
   0x0000b680:          4           1    <_fflush_r>
   0x0000bb14:          0           1    <_malloc_r>
   0x0000e5b0:          0           1    <__swrite>
   0x0000e704:          2           1    <__sclose>
```

```
   0x0000e72c:          0           1    <_free_r>
   0x0000efb8:          0           1    <_exit>


Data Cache Statistics:
                          total        hit       miss    miss-rate      reuse
   Reads             :        195        162         33          16%
   Bytes Read        :        547        428        119           -       0.52
   Writes            :        156         65         91          58%
   Bytes Written     :        523        231        292           -       0.10
```

References

[1] 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.

[2] Joseph A. Fisher, Paolo Faraboschi, and Young Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann (Elsevier), 2005.

[3] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th annual Workshop on Workload Characterization*, 2001.

[4] ISO. *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, 1990.

[5] ISO. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, 1999.

[6] 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.

[7] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[8] Peter Puschner. The single-path approach towards WCET-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pages 699–704, Dec. 2003.

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

[10] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

[11] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[12] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.