Project Number 288008

D 5.4 Report on WCET-Oriented Optimizations

Version 1.0
7 October 2013
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.	15 March 2013
0.2	First draft for review.	30 September 2013
0.3	Pre-final version, requesting partner comments.	4 October 2013
1.0	Final version	7 October 2013

Executive Summary

This document describes the deliverable *D5.4 Report on WCET-Oriented Optimizations* of work package 5 of the T-CREST project, due 24 months after project start as stated in the Description of Work. The deliverable describes the compiler optimizations that use the Patmos specific hardware features for time-predictable WCET improvement. We present the current state of the integration of the compiler and the WCET analysis, which enables more precise WCET analyses as well as WCET-driven code optimisations. We show how the feedback from the WCET analysis is used to perform data cache optimisations and criticality-driven control flow optimisations such as block placement and function splitting. Finally, we present code optimisations that are specific to the Patmos hardware, such as stack cache control optimisations and scheduling for the Patmos instruction set.

1 Introduction

Optimising compilers aim at producing efficient code, traditionally either in terms of time or space. Considering programs for hard real-time systems, optimisation targets are quite different. Time-predictability is of utmost importance, and the performance is captured by the results of static timing analysis rather than the average execution time measured for a set of benchmark executions.

In the initial project phase, we have identified several different aspects of time-predictability and the generation of time-predictable code (cf. D1.1). One aspect is *WCET analysability*, which is a measure of the complexity of the WCET analysis that reflects the effort required to produce a reasonably tight WCET bound.

From a hardware perspective, high analysability is achieved by avoiding a hardware design that employs features that are inherently difficult to analyse and by using easier-to-analyse alternatives instead. Often, these alternatives require appropriate support from the compiler. A representative example is a statically scheduled VLIW architecture instead of a superscalar processor with dynamic scheduling.

A different means to achieve high analysability is to provide the timing analysis with additional information. Even though the generated code is identical, the information leads to an improvement if the analysis is able to use it to obtain a tighter WCET bound. This sort of information is potentially available during compilation.

The flow of information in the other direction, namely from the analysis tool to the compiler, can be exploited to direct optimisations that aim at reducing the WCET. This includes information from value analysis and cache analysis, as well as information on the WCET of basic blocks, and their execution frequency on the worst-case path.

A major challenge for feedback-driven optimization, and the compiler and WCET analysis tool integration in general, is that the code potentially changes in each optimization step. As the compiler passes and the analysis tool thus operate on different program representations, mechanisms to transform analysis results are needed.

In this document, we present how we address these issues, the integration of our solutions in the compilation tool chain and an evaluation thereof. The rest of the document is structured as follows. Section 2 describes how we improved the integration of compiler and WCET analysis to provide additional information available in the compiler to the analysis tool. Section 3 outlines the optimisations based on feedback from timing analysis. Optimisations using Patmos-specific hardware features are dealt with in Section 4. Section 5 documents how we met the WP 5 requirements, and Section 6 concludes the discussion.

2 Compiler – WCET Analysis Integration

We already have described the basic strategy for the integration of the compiler and the WCET analysis in Deliverable D5.3. We introduced a tool called `platin` to facilitate the information exchange between the compiler and analysis tools. Compiler information, flow constraints and analysis results are stored in a uniform manner using the `PML` file format [10].

Within the scope of Task 5.3, we have extended and improved this integration. While the focus so far was on *enabling* WCET analysis within our compiler framework, we now have focussed on *improving* the analysis results by providing more information to the analysis, both quantitatively and qualitatively.

In this section, we first outline the technique used to relate different program representations. Then we describe two kinds of information from the compiler which are processed and provided to the timing analysis, evaluating their effects on the analysis results.

2.1 Relating Bitcode and Machinecode

During compilation, programs are transformed from source code to the compiler's platform-independent intermediate language, and subsequently to the target platform's machine representation and the linked binary file. Bitcode, LLVM's platform-independent intermediate representation, was designed with analyzability in mind and has thus become an attractive representation for program analyses (e.g., [13]). As WCET analysis targets platform-specific machine code, however, it cannot directly reuse results from program analyses operating at the bitcode level.

Therefore, a major challenge for the integration of compiler and WCET analysis is to establish a link between different representations of the same program. Establishing such relations is not only crucial for the reuse of existing knowledge about the program, but also helpful to use WCET analysis results to guide optimizations (see Section 3). We tackled this problem by developing a novel representation called control-flow relation graphs (CFRGs)[6], which provide an accurate model of the control-flow relation between machine code and the compiler's intermediate representation. By modeling the combined control-flow representations as an integer linear program, and applying a variant of the well-known Fourier-Motzkin elimination algorithm [12], it is possible to automatically transform any linear flow constraint from one representation to the other, in a sound and precise way. As a consequence, we are able to reuse the results from LLVM's *Scalar Evolution* analysis, a symbolic value analysis already used for optimizations on the bitcode level, for WCET analysis (see Section 2.2).

In order to facilitate the integration of CFRGs in LLVM, we developed a construction algorithm that builds control-flow relation graphs from partial block mappings, and modified LLVM to ensure these mappings stay valid during backend transformations. The necessary modifications are minimal, as we reuse LLVM's existing and largely platform-independent mechanisms to keep track of block relations. The evaluation of this technique for the Patmos processor revealed that it facilitates the precise transformation of flow information.

In the following, we provide an overview of CFRGs and relevant algorithms, and illustrate the technique by means of an example. Furthermore, we present evaluation results obtained using the current versions of the compiler and WCET analysis tool. For further details on algorithms, formal definitions and proofs, see [6].

2.1.1 Control-Flow Relation Graphs

A CFRG $\hat{\mathcal{G}}$ is a data structure that represents regular path relations between two CFGs $\mathcal{G}_a$ and $\mathcal{G}_b$. Similar to the way a CFG models a set of execution paths, CFRGs model a set of execution path *pairs*.

The set of nodes of a CFRG $\hat{V}$ is partitioned into three different kind of nodes. *Progress nodes* ($\hat{V}_p$) ensure that progress in one representation is matched by progress in the other one, and are labelled with a pair of basic blocks from $\mathcal{G}_a$ and $\mathcal{G}_b$. The progress nodes $\hat{s}, \hat{t} \in \hat{V}_p$ are the unique entry and exit nodes without predecessors and successors, respectively, and correspond to the respective entries and exits of $\mathcal{G}_a$ and $\mathcal{G}_b$. In addition to progress nodes, a CFRG comprises *source nodes* $\hat{V}_a$ and *target nodes* $\hat{V}_b$, that correspond to the execution of one basic block in either representation. Source nodes are labeled with one block from $\mathcal{G}_a$, target nodes with one block form $\mathcal{G}_b$. Edges in the CFRG are either source edges $\hat{E}_a \subseteq (\hat{V}_p \cup \hat{V}_a) \times (\hat{V}_p \cup \hat{V}_a)$, or target edges $\hat{E}_b \subseteq (\hat{V}_p \cup \hat{V}_b) \times (\hat{V}_p \cup \hat{V}_b)$. Edges from $\hat{E}_a$ and $\hat{E}_b$ correspond to a change of control in the functions modeled by $\mathcal{G}_a$ and $\mathcal{G}_b$, respectively.

The intuitive idea of the CFRG representation is the following: assume π_a and π_b are two related paths. Then the respective first nodes u_a and u_b and last nodes v_a and v_b of π_a and π_b should be related as well, and correspond to progress nodes $\hat{u}$ (labeled with (u_a, u_b)) and $\hat{v}$ (labeled with (v_a, v_b)). In case there is no (modeled) relation between subpaths of π_a and π_b, there will be a path of $\hat{E}_a$-edges corresponding to π_a, and a path of $\hat{E}_b$-edges corresponding to π_b, such that no interior node of $\hat{\pi}_a$ and $\hat{\pi}_b$ is a progress node. Conversely, if there are prefixes of π_a and π_b that are related, then there will be some progress node $\hat{w}$, labeled with the last nodes of the related prefixes. Now the same considerations apply recursively to the related prefixes (from $\hat{u}$ to $\hat{w}$), and the related suffixes (from $\hat{w}$ to $\hat{v}$) of π_a and π_b. Formally, this intuition is made precise by means of regular path relations [6]; in this report, we restrict ourselves to an illustrative example, presented next.

Example 1. *Figure 1 shows two different CFGs and the CFRG for a function that counts the number of digits and whitespace in a string. For simplicity, the semantics of each block's instructions is illustrated using a C-like pseudo language. Figure 1a shows the CFG of the function at the IR level. Figure 1b represents the same program after a few typical backend transformations have been performed: (1) due to layout optimizations, a new basic block is introduced at the entry of the function (2) the switch instruction is replaced by two conditional branches; as a consequence, one outgoing edge of the switch block corresponds to two paths in the transformed program (3) the branch to the basic block* `digit` *is eliminated using a predicated instruction (if-conversion) (4) the tail of the loop body is duplicated.*

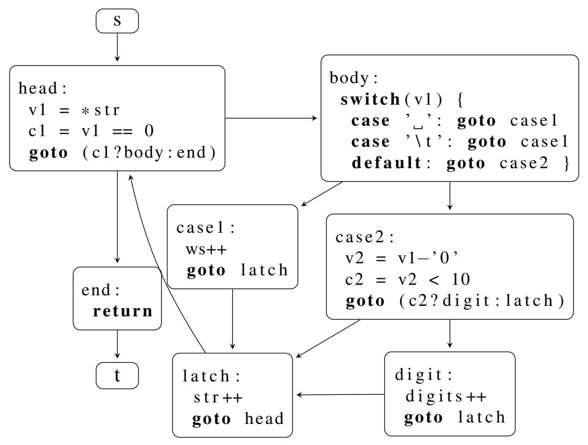
Figure 1c visualizes the control-flow relation graph for this example. There are eight progress nodes, each labeled with a pair of CFG nodes, four source nodes and two target nodes. One relation modeled by this graph is the one between the IR-level path

$$\pi_a = \texttt{body} \to \texttt{case1} \to \texttt{latch} \to \texttt{head}$$

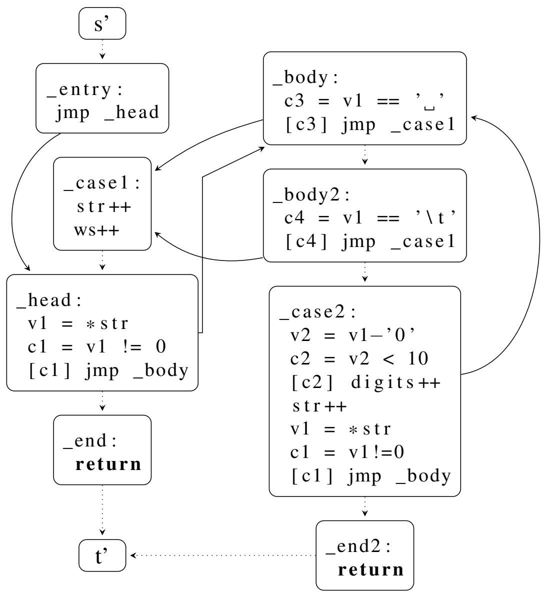
and the two machine-code paths

$$\pi_b^1 = \texttt{\_body} \to \texttt{\_case1} \to \texttt{\_head} \quad \pi_b^2 = \texttt{\_body} \to \texttt{\_body2} \to \texttt{\_case1} \to \texttt{\_head}$$

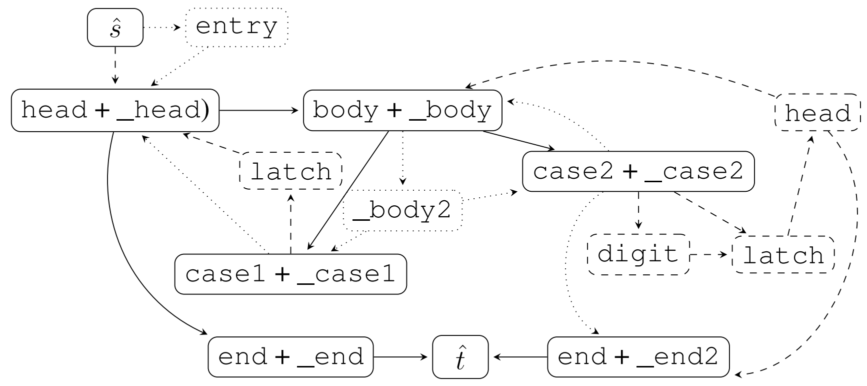
Indeed, inspecting the respective CFGs, we see that every time the sequence π_a *is executed at the IR-level, either* π_b^1 *or* π_b^2 *is executed on the machine-code level.*



(a) IR-level CFG



(b) Machine-code–level CFG



(c) Control-Flow Relation Graph

Figure 1: Illustration of two different CFGs and the CFRG of a function counting digits and white spaces in a string. In the machine-code representation, dashed edges are fall-through transitions; predicates guarding the execution of an instruction are written in square brackets. In the CFRG, progress nodes are drawn with solid, source nodes with dashed and target nodes with dotted shapes. Dashed lines correspond to source edges, dotted lines to target edges; we draw solid lines if there is both a source and target edge.

2.1.2 Flow Fact Transformation

The CFRG representation allows to transform control-flow related information from one representation to the other. In particular, we have implemented a technique to transform flow facts from the IR level to machine code in a sound way (see Section 2.2).

The transformation of flow facts proceeds as follows. First, we build integer linear programs that model the control-flow in the IR representation, and in the machine code representation, respectively. In these models, which are build using the IPET technique [11], variables correspond to the execution frequency of control-flow edges. Next, we derive equations from the control-flow relation graph which relate variables from both representations. These equations relate edges in the CFRG with edges in one of the CFGs, assert that flow is preserved for CFRG edges, and express the fact that flow through progress nodes is the same in both representations.

The integer linear program obtained this way contains edges from both CFGs, as well as CFRG edges. In order to derive flow facts on the machine-code level, we add the IR-level flow constraints to the model, and then eliminate all CFRG edges and edges from the IR-level CFG using our implementation of the Fourier-Motzkin elimination algorithm. An edge e is eliminated as follows: if e appears in an equation $e = T$, replace all occurrences of e by T. Otherwise, partition the set of constraints where e appears into a set of lower bounds C_L of the form $T_L \leq e$ and a set of upper bounds C_U of the form $e \leq T_U$. Then add the constraints $T_L \leq T_U$ for all $T_L, T_U \in C_L \times C_U$, and remove the original constraints and variable e from the problem. In order to improve the efficiency of this procedure, we eliminate all edges at once, and use a heuristic to guide the selection of variables to be eliminated next.

2.1.3 Construction Algorithm and LLVM Integration

The effort to keep a CFRG-based relation model up-to-date in the backend might be challenging even for compilers engineered with CFRG support in mind, and even more so for existing industrial-quality compiler frameworks such as LLVM. Therefore, we developed an algorithm that allows to build CFRGs from partial block mappings, which need considerably less effort to be integrated into a compiler backend. This is in particular true for LLVM, as we could reuse and adapt existing infrastructure for maintaining basic block relations, and then apply the construction algorithm.

From the perspective of the CFRG construction algorithm, the compiler needs to provide a *partial* mapping from basic blocks (both at the bitcode and machine code level) to events. This mapping needs to satisfy the property: assuming that every basic block emits the event it is associated with (if any), and the function is executed in both representations with equivalent input, the same sequence of events is emitted in both program representations. Given that the event mapping provided by LLVM is correct, the construction algorithm is guaranteed to provide a correct CFRG.

In our implementation, the set of events is a subset of IR-level basic blocks. In the compiler, we maintain a partial mapping $\mathcal{B}$ from machine basic blocks to IR-level basic blocks. If a bitcode basic block B_a is in the range of $\mathcal{B}$, it is associated with the event $e(B_a)$; otherwise, no event is assigned to B_a. For machine-code basic blocks, if B_b is in the domain of $\mathcal{B}$, it is associated with the event $e(\mathcal{B}(B_b))$; otherwise no event is assigned to B_b. Thus every machine-code block that is in the domain of $\mathcal{B}$ is associated with the same event as the corresponding bitcode-level block.

Minor modifications to ensure that the block mapping $\mathcal{B}$ matches the requirements of the CFRG construction algorithm were necessary as well. Most notably, the update of $\mathcal{B}$ during the *tail merging* optimization had to be modified. This optimization pass introduces a new machine basic block which corresponds to suffixes (tails) of two different blocks before the optimization. As $\mathcal{B}$ associates a machine basic block with at most one bitcode basic block, $\mathcal{B}$ has to be undefined for these newly introduced blocks.

In our toolchain, CFRGs are integrated as follows. At the end of the code-generation phase we export both the IR-level CFGs and the machine-code level CFGs, as well as the CFRGs obtained using the construction algorithm using the `PML` format. The `platin` tool subsequently uses CFRGs to transform flow information to be used during WCET analysis; the LLVM compiler relies on relation graphs to interpret analysis information for optimizations.

2.1.4 Evaluation

The goal of the evaluation presented here was to assess whether CFRGs provide precise control-flow relations, and to obtain insights on the size of CFRGs and the number of constraints generated by our flow-fact transformation technique.

The baseline for the evaluation is the WCET bound calculated using a set of predetermined flow facts at the machine code level. This bound is compared with the WCET analysis result obtained using our flow-fact transformation technique and flow facts on the bitcode level. In order to ensure a fair comparison, we did not generate bitcode-level flow facts with a different tool, but transformed the original set of flow facts to bitcode (roundtrip).

We attempted to use a wide variety of flow facts in the evaluation, that should reference all basic blocks (and not just loop headers, for example). An effective way to obtain such flow facts for evaluation purposes is to use flow-fact generation from execution traces, a feature that has been implemented in the `platin` tool. Note that for deterministic benchmarks, there is exactly one feasible execution path, and thus the flow facts extracted from machine-code traces are indeed precise characterizations of the benchmark run.

The evaluation proceeds as follows: First, each benchmark is compiled using the Patmos port of LLVM, resulting in the binary and the PML file, that includes the CFRGs for all functions. We use the default optimization level (-O2), which enables the optimizations tail duplication, branch folding, tail merging and if-conversion; all of them modify the structure of the CFG. Besides optimizations, differences in the structure of the CFGs stem from the fact that some bitcode instructions require the compiler to introduce additional basic blocks. A notable example is the switch-statement, which is lowered in different ways, depending on the number of cases as well as the range and the density of the case label values.

The WCET calculation itself is carried out twice for each binary, once with the set of flow facts obtained from the trace analysis, and second with the set of transformed flow facts. As the evaluation deals with WCET calculation, not low-level timing analysis, we settled for a simple configuration that assumes fast, local memories. Different low-level timing models would only change the cost model that is used for both WCET estimates.

Table 1: Evaluation of CFRGs

Benchmark	$\|V_a\|$	$\|V_b\|$	$\|u(V_b)\|$	$\|\hat{V}\|$	T_b	$\frac{\hat{T}-T_b}{T_b}$
Mälardalen WCET benchmarks						
adpcm	37	35	11.43%	49	2699576	0.00%
bsort100	11	13	15.38%	15	126353	0.00%
cnt	34	5	20.00%	36	3359	0.00%
compress	50	50	16.00%	65	5345	0.00%
cover	16	19	0.00%	20	1122	0.00%
crc	7	9	22.22%	11	14364	0.00%
edn	34	37	13.51%	42	93077	0.00%
expint	11	9	0.00%	14	92189	0.00%
fdct	4	5	20.00%	6	2206	0.00%
fft1	172	176	14.77%	229	62806	0.34%
jfdctint	5	7	28.57%	8	4064	0.00%
lcdnum	21	22	0.00%	23	201	0.00%
lms	358	338	13.31%	479	6931095	0.16%
ludcmp	189	197	13.71%	240	66195	0.05%
matmult	13	15	13.33%	16	162065	0.00%
minmax	11	7	0.00%	13	93	0.00%
minver	191	193	14.51%	249	12110	0.35%
ndes	45	26	7.69%	49	43744	0.00%
ns	11	12	8.33%	13	8130	0.00%
nsichneu	753	501	0.00%	754	7918	0.00%
qsort	54	46	13.04%	70	4097	2.27%
qurt	192	193	15.03%	256	31941	0.32%
select	56	48	16.67%	76	4840	1.74%
statemate	15	12	8.33%	17	197	0.00%
ud	105	114	12.28%	130	14500	0.14%
other	≤ 10	≤ 10	0%	≤ 10	-	0.00%
PapaBench WCET benchmark (Tasks)						
check_failsafe	120	124	12.10%	152	1477	0.00%
check_mega128_values	122	124	11.29%	152	1480	0.00%
send_data_to_autopilot	108	107	12.15%	135	666	0.00%
servo_transmit	93	71	0.00%	98	504	0.00%
test_ppm	245	236	10.17%	308	4714	0.00%
altitude_control	89	78	10.26%	120	319	0.00%
climb_control	232	220	11.82%	301	1931	0.31%
link_fbw_send	6	3	0.00%	8	42	0.00%
navigation	633	500	15.40%	902	7967	0.62%
radio_control	285	252	11.11%	362	42	0.00%
receive_gps_data	374	357	12.04%	495	51	0.00%
reporting	108	89	3.37%	115	521	0.00%
stabilisation	201	187	12.83%	272	1289	0.70%

We investigated benchmarks from two suites popular in the WCET community; 33 benchmarks from the MRTC benchmark set, and the real-time tasks from the PapaBench WCET benchmark [1]. Table 1 summarizes the results of the evaluation. $|V_a|$ and $|V_b|$ give the total number of basic blocks in the bitcode and machine code CFGs, respectively. The column $|u(V_b)|$ shows the percentage of unmapped machine-code basic blocks, that is, the percentage of blocks not in the domain of $\mathcal{B}$. $|\hat{V}|$

[1]All sources are available as part of our benchmark collecting at `https://github.com/t-crest/patmos-benchmarks`

corresponds to the total number of relation graph nodes. In the column labeled with T_b, the WCET using the original set of machine-code flow facts is given. The column labeled $(\hat{T} - T_b)/T_b$ shows the increase of the calculated WCET bound when using transformed flow facts, relative to T_b.

The percentage of unmapped machine-code basic blocks, that is, those blocks not in the domain of $\mathcal{B}$, is an indicator for the (local) uncertainty in the initial control-flow mapping provided by the compiler. Although in theory the size of a CFRG need not be linear in the number of basic blocks, in the evaluation it turned out to be at most twice as large.

Analysis tools that operate on the bitcode level are portable and operate on a program model that provides more information than a binary file. As our CFRG model relies on control-flow abstractions only, the transformation of IR-level flow facts to machine code potentially looses information needed to calculate a precise WCET bound. In our evaluation, however, we found that in all of the benchmarks investigated, there is no substantial loss of precision due to the transformation process.

2.2 Symbolic Flow Fact Integration

Optimising compilers provide transformation passes that require some form of program analysis prior to the actual transformation. Although information from such analyses would be valuable also for timing analysis, it is usually discarded after the transformation pass requiring it. This section provides one example to demonstrate the benefits of the availability of information from the compiler for the WCET analysis phase.

Recent research efforts have focused in exploiting the properties of a type-enriched SSA representation, like the bitcode IR of LLVM, to generalise and improve compiler analyses and transformations that were formerly tied to a source level representation [9]. The benefits of such analyses on this high-level intermediate representation are evident: They are independent of both source language and target platform. In addition, the usage of symbolic names and values provides a high degree of accuracy, as opposed to the lower-level machine code owing to concrete register names and sub-word operations.

In order to be WCET-analysable, maximum iteration counts for loops (*loop bounds*) must be known statically for programs. Although the timing analysis tool `aiT` features a loop bound analysis, the low-level nature of the machine code often prohibits computation of loop bounds solely with the information available in the binary. As a consequence, the programmer is requested to provide bounds manually – a task that is both tedious and error-prone.

However, often loop bound information is readily available in the compiler. The LLVM framework features a loop induction variable analysis that computes the evolution of program values during loop iterations, in order to reduce computation costs within the loop (*loop strength reduction*). This bitcode analysis pass named *Scalar Evolution* computes closed-form expressions for maximum backedge-taken counts of predictable loops, such that extraction of loop bounds from the intermediate program representation becomes possible.

Loop iteration counts often depend on the program state or the execution context. In the following, we will distinguish between two kinds of loop bounds:

Constant loop bounds.
They are valid in all execution contexts of the corresponding loop.

Symbolic loop bounds.
They are parametrized over program or register values at a specific program point.

The goal of the symbolic flow fact integration is to provide as much loop bound information as available automatically to the analysis tool. This way, the burden on the programmer to provide loop bounds manually should be reduced. Furthermore, the quality of the bounds should be high to achieve a precise analysis result. To this end, we support symbolic loop bounds that are parametrised over formal function arguments.

Example 2 (Running example). *Consider the example in Figure 2. The C source code fragment describes a function containing a nested loop. The iteration count of the outer loop depends on function parameter* `ub`*. Furthermore, the iteration count of the inner loop depends on the value of* `i`*, the induction variable of the outer loop, a property defining a so-called* triangle loop.

```
void f(int ub) {
    int i,j;
    for(i = 0; i < ub+1; i++) {
        for(j = 1; j < i; j++) {
            work(i,j);
        }
    }
}
```

Figure 2: A triangle loop for which the loop iteration count of the outer loop depends on the function parameter `ub`.

The Scalar Evolution analysis provided by the compiler is centered around the concept of *chains of recurrences* [14]. The idea is to represent loop-variant values as recurrences on the trip count of loops. Recurrences supported by LLVM and thus by our analysis take the form $s+ci$, where i is the iteration counter of a loop (counting from 0 up with increment 1), s is the (loop-invariant) start value and c is the loop-invariant stride. LLVM's Scalar Evolution analysis computes loop bounds by inspecting exit conditions, and analyzing comparisons between the chain-of-recurrences representation of the loop induction variable and loop invariant variables. The loop bound expressions computed reference bitcode variables and, by means of chains of recurrences, the iteration counter of outer loops.

Figure 3 illustrates the loop bound expressions computed by Scalar Evolution for the example in Figure 2. The outer loop's body is executed `ub+1` times every time the outer loop is entered, or not at all if `ub` is less than one. The bound for the inner loop depends on the iteration count of the outer loop. In the first two iterations of the outer loop, the inner loop's body is not executed at all, in the i^{th} iteration of the outer loop, it is executed $i - 1$ times.

```
Backedge count inner loop:  max(0,{-1 + i}<outerloop>)
Backedge count outer loop:  max(%ub + 1, 0)
```

Figure 3: Scalar evolution expressions for the backedge counts of the triangle loop

Formula	A	Condition
$(L' - A)s + \frac{L'(L'-1)-A(A-1)}{2}c$	$\max(0, \lfloor\frac{-s}{c}\rfloor)$	$c \geq 0 \wedge A < L'$
0	$\max(0, \lfloor\frac{-s}{c}\rfloor)$	$c \geq 0 \wedge A \geq L'$
$As + \frac{A(A-1)}{2}c$	$\max(0, \min(\lfloor\frac{s}{-c}\rfloor, L' - 1)$	$c < 0$

Table 2: Total trip counts of triangle loops from Scalar Evolution

Loop bounds that depend on the iteration counter of the outer loop need to be resolved to obtain flow facts that can be interpreted by the WCET analysis tool. A loop bound expression $L(i) = \max(0, s + ci)$, where c is a integer constant and i is the iteration counter of an ancestor loop, either increases or decreases in a monotone way, depending on whether c is positive or negative. The expression $L(i)$ thus takes it maximum at $\max(0, s)$ if c is negative. If c is positive, and L' denotes the (possibly symbolic) loop bound of the outer loop, the inner loop's bound is $\max(0, s+c(L'-1))$.

These loop bounds are not always sufficient for precise WCET estimates, however, as the maximum loop bound depends on the trip count of the ancestor loop. Therefore, we also extract bounds for the frequency of the loop header relative to the entry of the outer loop.

The bound on the number of times the inner loop's header is executed relative to the ancestor's loop entry is

$$B = \sum_{i=0}^{L'-1} \max(0, s + ci)$$

The only complication in this formula is the maximum inside the sum, and the fact that L' need not be positive. As noted in [5], this is the reason why simply taking the well-known closed form for the arithmetic series is not correct in the general case. Fortunately, the maximum be eliminated by splitting the sum in two parts (this in turn is possible as $s + ci$ is monotone). Table 2 summarizes the formulas for the total trip count of triangle loops. The second and first column show a helper variable (A) and the formula, which is applicable if the condition in the third column holds.

This kind of loop analysis technique has been applied in the context of timing analysis before [5]. The most compelling advantage of our tightly-integrated solution is that we first extract *symbolic loop bounds* that are later on instantiated (in the *aiT* tool) using values obtained by context-sensitive abstract interpretation. Moreover, our integration strategy enjoys long-term benefits from direct compiler integration; any improvements in the compiler, or the integration of other bitcode analysis tools, will also improve the analysis.

Example 3. *In the example program from Figure 2, the outer loop bound is* $\max(0, ub + 1)$*, and the inner loop bound* $-1 + i$*. Therefore, the closed form for the inner loop bound is* $\max(0, -1 + (\max(0, ub + 1)) - 1)$*, which simplifies to* $\max(0, ub - 1)$*. Indeed, as* i *is at most* ub *(in the last iteration), the inner loop is executed at most* $ub - 1$ *times, or not at all if* $ub \leq 1$*. The bound for executions of the inner loop's body relative to the function entry is obtained using the formula for* $c \geq 0$*. In this case, we have* $c = 1$ *and* $s = -1$*, and so* $A = 1$*. The formula then simplifies to*

$\frac{ub(ub-1)}{2}$. *Also considering the case when* $ub \leq 1$, *the inner loop's body is thus executed at most* $B = \max(0, \frac{ub(ub-1)}{2})$ *times for every execution of the function.*

In the information export phase, implemented in LLVM, support for symbolic flow facts consists of two parts. First,the scalar evolution pass is queried. If a scalar evolution expression describing the backedge count of a loop is either constant or parametrised over formal function arguments only, the expression is exported as flow fact refering to the bitcode representation to the PML database. Second, a mapping of the symbolic names of function arguments to machine registers, which will eventually hold the respective values, is exported for machine functions if the argument is an integer type and contained in machine registers. This information is target dependent and attached to functions of the machine code level representation.

Symbolic loop bounds are communicated to the WCET analysis tools by transforming them from the bitcode to the machine code level, and subsequently generating appropriate annotations for `aiT`. In `platin`, we parse the symbolic bounds and resolve affine chains of recurrences as discussed above. Next, we use the control-flow relation graph as well as the argument register mapping to transform the loop bound from bitcode to the machine code level. Figure 4 shows the resulting flow constraints for the running example.

```
Backedge count outer loop  (MC): max(r3 + 1, 0)
Backedge count inner loop  (MC): max(0, max(r3+1,0)-2)
Total Backedge count inner (MC): max(r3+1,0)*(max(r3+1,0)-1)/2
```

Figure 4: Before export to aiT, we resolve chains-of-recurrences and transforms symbolic flow facts from bitcode to machine code.

The `aiT` tool allows to specify loop bounds that depend on so called user variables. These variables exist at the analysis level only, and may be assigned to expressions involving registers, at any instruction in the program. We exploit this mechanism by assigning the value of argument registers to user variables at function entries, and then reference these user registers in the symbolic expression specifying the loop bound (see Figure 5).

```
instruction "f" is entered with @arg_r3 = trace(reg r3);
    # extracted argument for symbolic loop bound
loop ".LBB2_2" max (@arg_r3 + 1, 0) end;
    # global loop header bound (source: llvm)
```

Figure 5: AIS annotations for the timing analysis tool, describing the parametric loop bound of the inner loop.

2.3 Providing Information on Memory Addresses

To aid the timing analysis tool in the loop and value analysis phase, we have extended the compilation tool chain to provide information about targets of memory accesses (cf. Deliverable D6.1, Section 3.2). By providing this additional information, we expected an improvement in the quality of the timing analysis result. The auxiliary information can be used to improve the precision of the data cache analysis. Furthermore, it is the basis for an optimisation that exploits the memory organisation of Patmos and leads to tighter WCET bounds (see Section 3.2).

The timing analysis tool accepts memory access targets by means of AIS annotations of the form

`instruction` *address* `accesses:` *expression* `;`

where *expression* provides the target of the memory access by means of the accessed symbol and an optional offset.

2.3.1 LLVM Implementation

We remind the reader that the the LLVM intermediate language (bitcode) is in *Single Static Assignment* (SSA) form, i.e., a variable is assigned exactly once. To facilitate assignments of variables from different sources of input resulting from control flow, SSA features the ϕ function, which "chooses" the right definition from its operands [3]. Futhermore bitcode features `load` and `store` instructions that have a pointer operand specifying the address of the memory access. In the LLVM backend framework, memory accessing machine instructions can be equipped with a *memory operand*, which references the pointer operand of the bitcode representation.

During the information export phase, whenever a memory-accessing machine instruction is encountered, the value of the memory operand is read. Then we employ an algorithm operating on the bitcode representation starting from the pointer value as described below. If the algorithm is able to narrow down the memory accesses to static data to one or more symbols, this information is exported as a *value fact* refering to the respective machine instruction to the PML database, processed by platin and eventually translated to a AIS annotation for the timing analysis tool.

In our implementation, we try to expose memory accesses to global symbols that refer to compound data objects (arrays, records). Elements of these data objects are addressed by the LLVM `getelementptr` instruction (GEP), that has a base address (with type) as first operand and a list of indices that specify the exact location of the item within the compound structure.

In the simplest case, the base address is a global symbol. But it could also be a computed value, e.g., by another GEP instruction. Or it could result from a ϕ operator, the merge-operator of the SSA-form. The algorithm we use to obtain possible memory targets is described in Algorithm 1. It collects symbols which are possibly accessed in a set *symbols*. The value the memory operand of the machine instruction is refering to is inspected. If it is an access to a global symbol (by GEP), the symbol is added to *symbols* and the algorithm terminates. Otherwise, if the base address is computed, the computed value is inspected. If the value is a ϕ operator, all its operand values are inspected. A value is visited at most once: each time a value is inspected, it is added to a set of already visited values, to ensure termination in case of circular definitions, as it is common for values defined in loops. In all other cases, i.e., the value is neither a GEP nor a ϕ operator, a $\top$ is added to *symbols*. After the algorithm has terminated, if the set of possibly accessed symbols contains $\top$, no precise information about accessed memory has been obtained. Otherwise, the memory areas referenced by *symbols* are exported as possible memory access targets.

Algorithm 1 Collecting possibly accessed global symbols.

```
global $visited = \emptyset$, $symbols = \emptyset$

procedure EXPLORE($v$):
    $visited = visited \cup \{v\}$
    case $v.kind$ of
    GEP:
        if indexesGV($v$)
            // Base case
            $symbols = symbols \cup \{v\}$
        elseif $v.baseptr \notin visited$
            EXPLORE($v.baseptr$)
        return
    PHI:
        foreach $w \in v.operands$ do
            if $w \notin visited$
                EXPLORE($w$)
        return
    default:
        $symbols = symbols \cup \{\top\}$
        return
```

2.3.2 Evaluation

We compared the quality of the timing analysis results obtained with no address information to the results with address information available. The benchmarks consisted of the Mälaradalen benchmarks, tasks from PapaBench and three synthetic benchmarks.

As for the data cache configuration, we assumed a 4-way set-associative data cache with 2KB size and a cache line size of 32 Bytes. The transfer time consists of a constant request time and the transfer time for each successive double-word. The request time was assumed as 10 cycles, and 4 cycles for each successively transferred double-word. Loading one double-word from main memory thus takes 14 cycles, a cache line of four double-words takes 26 cycles. See [7] for a rationale for these numbers.

The number of exported value facts for memory addresses, which corresponds to the number of instructions for which address information was available, is given in Table 3. Benchmarks, for which no information was exported at all, are not listed in the table.

While the information had a very little effect on the computed worst-case execution time bound, we could observe changes in the aiT classification of memory accesses. The tool puts each access into one of four categories: *exact*, *nearly precise*, *imprecise*, and *unknown*. An access is exact if the address of the memory access is known. An access is imprecise if the access is known to be in a

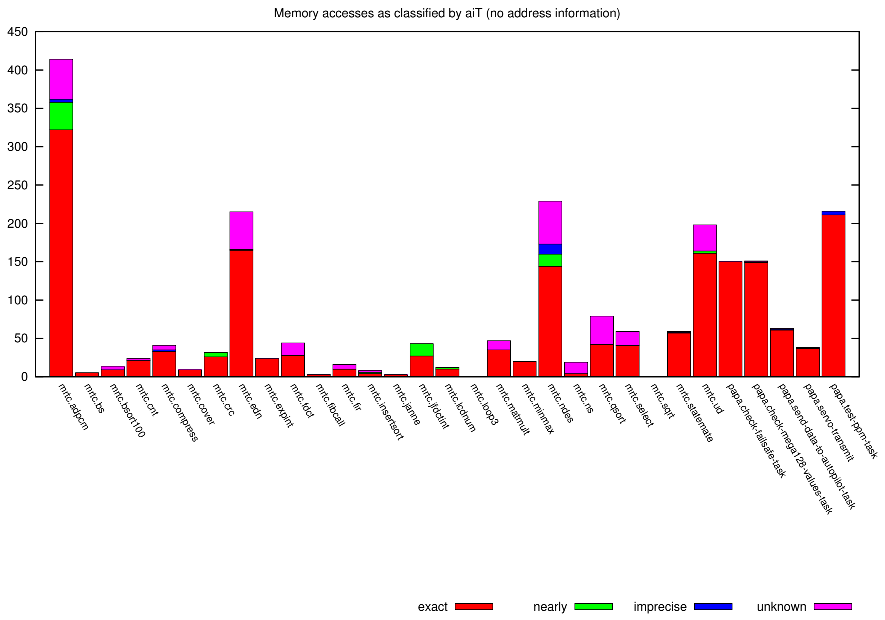
benchmark	-O0	-O1	-O2
mrtc_adpcm	1	24	1
mrtc_bs	3	0	0
mrtc_bsort100	0	1	1
mrtc_cnt	0	0	2
mrtc_compress	1	6	6
mrtc_crc	15	2	1
mrtc_fdct	0	2	2
mrtc_insertsort	6	0	0
mrtc_jfdctint	1	3	3
mrtc_matmult	0	7	7
mrtc_ndes	18	12	9
mrtc_ns	3	0	0
mrtc_nsichneu	378	1	1
mrtc_qsort	43	15	15
mrtc_select	39	17	16
mrtc_ud	1	0	0
papa_fbw	12	17	2

Table 3: Number of exported value facts for memory accesses, for the benchmarks with different optimisation levels.

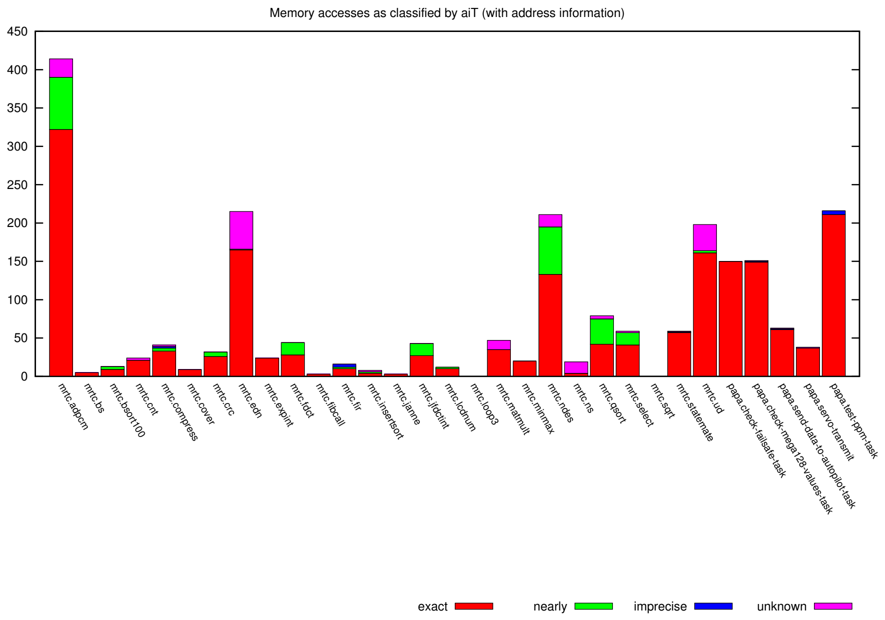
.

given range, if the range is below a certain threshold (1024 Bytes), the access is classified as nearly precise.

Figure 6 depicts the impact of the address export on the memory access classification by aiT, on memory reads with the benchmarks compiled with basic optimisations enabled (`-O1`). We observe that with our export strategy the number of precise accesses remains unchanged. For benchmarks where address information could be obtained and exported from the compiler (e.g., adpcm, ndes, qsort, select), accesses could be classified more precisely, i.e., accesses that were classified as unknown before, are classified as imprecise or nearly precise. Also, the range of imprecise accesses could be narrowed such that the accesses became nearly precise.



(a) Memory access classification without address-export



(b) Memory access classification with address-export

Figure 6: Impact of the address export on the memory access classification by aiT.

2.4 Stack Cache Related Information

In Deliverable D5.3 we described the compiler support for the stack cache that is employed in Patmos. The stack cache allows the program to store fixed size stack frames locally at the processor node. The Patmos ISA provides special instructions, which are used to reserve and to free space in the stack cache, as well as to load stack frames back from memory after possible eviction by callees. Costly memory transfers to global memory are limited to those special instructions.

In order to reduce the number of generated stack cache control instructions as well as the amount of memory filled and spilled by the stack cache, the compiler performs a stack cache optimisation using a stack cache occupancy analysis. For a description of the analysis and the optimisation, we refer to Section 4.2.

The results of the occupancy analysis are not only used to perform the stack cache optimisation, but also to aid the WCET analysis tool with the static analysis of the stack cache related spill and fill costs. Based on the maximum cache displacement found by the stack cache occupancy analysis, the compiler derives worst case values for the maximal number of blocks spilled or loaded from main memory at stack control instructions **sens** and **sres**.

We use the PML export framework to emit those numbers for all stack control instructions in the application code. They are then exported to aiS annotations by the `platin` tool.

3 Feedback-directed WCET Optimisations

In the previous section, we focussed on the flow of information from the compiler to the WCET analysis, which is used to improve the precision of the WCET bound. In this section we present optimisations that use information from the WCET analysis inside the compiler, thus closing the loop of the compiler and WCET analysis integration. We will first discuss the challenges in back-annotating WCET analysis results to compiler passes and present our approach to back-annotation and feedback-driven optimisation, and then present optimisations that make use of the information from the WCET analysis in order to reduce the WCET of the application tasks.

3.1 Introduction

WCET-guided optimisations attempt to use information provided by WCET analysis tools, such as `aiT`, to guide the optimisation process. In general, optimisation passes operate on a program representation different from the one that is reflected in the final executable, due to the effect of subsequent optimisations. This in turn complicates the flow of information from the analysis tool to the optimisation pass, and requires to interpret the analysis information in the context of the optimisation's input.

In order to avoid this problem, it seems tempting to analyze the WCET of the code that it is fed into the feedback-directed optimisation pass, instead of the final executable. However, WCET analysis tools need to operate on linked binaries, as timing is only defined if all machine code with address information is available.

Instead of analysing the input to the optimisation pass directly, one might disable subsequent optimisations, generate a linked executable, analyse it and use the results to guide the optimisation. This simplifies the processing of the analysis information, as the analysed program is closer to the one subject to optimisation. The downside of this approach is, however, that disabling all subsequent optimisations might lead to quite different analysis results, which in turn misleads the optimisation pass.

For example, consider the if-conversion optimisation, that is scheduled as one of the last optimisations in the Patmos backend. If it is disabled, it is indeed much easier to relate the final executable with the input to earlier optimisation passes. However, if-conversion also has a significant effect on the VLIW scheduler (as it results in larger basic blocks) and on the WCET analysis (e.g., instruction cache analysis is more precise for large basic blocks).

The approach we took within our framework actively deals with the problem of relating program information. We use the analysis results obtained for the fully-optimised program, and in turn accept the imprecision that stems from not being able to relate program representations precisely. This is similar to the way profile-guided optimisations are performed, although we also have to deal with more fine-grained analysis results, such as the address ranges determined for memory accesses.

The key ingredient of our solution is to keep a relation between the program before optimisations are run, the analysed program representation, and the program representation subject to optimisation (see Section 2.1). To this end, we implemented a generic framework for back-annotation in both `platin` and LLVM. Before every optimisation pass that uses information from WCET analysis, the

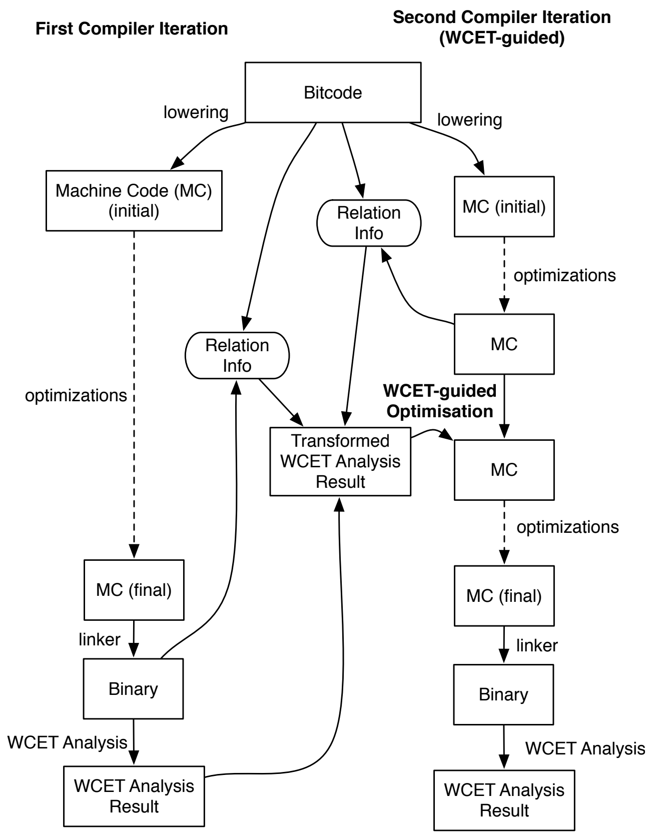


Figure 7: Strategy for WCET-guided optimisation

information is translated to program representation subject to optimisation (see Figure 7).We do not, however, rerun the WCET analysis tool before every optimisation step, reducing the number of costly analysis iterations. Another advantage of our approach is that analysis results from different sources can be processed; it is also possible to use information obtained from analyzing execution traces, for example.

In general, information from the WCET analysis tool is processed as follows. First, we interpret the information from the WCET analysis tool with respect to the program model used in `platin` (which in turns corresponds to LLVM's program model, extended with context sensitivity). The WCET analysis tool `aiT` treats loops as functions, and thus loop entry and exit edges are modeled by function calls and returns. This is a challenge when interpreting the analysis results, as call contexts are potentially merged when entering a loop. Next, the analysis results are written to the PML database. In order to perform WCET-guided optimisations, this PML file has to be provided to LLVM. Optimisation passes that need WCET analysis results use a generic query interface, that provides the desired results in the context of the optimisation passes' input.

In order to simplify the use of feedback-driven optimisations, we also provide a drop-in replacement for the C compiler (`patmos-clang-wcet`). The user needs to pass a configuration to this tool, that provides sufficient information to configure both compiler and WCET analysis, depending on the hardware configuration. Using this information, the WCET analysis tool is run automatically after compilation, and the results are fed back into the compiler in the subsequent iteration.

3.2 Value Analysis Feedback: Bypass Optimization

It is not always possible to determine the exact address or address range of a memory access during value analysis. This is troublesome, as an access where the address is not known invalidates large parts of the knowledge accumulated about cache contents. For a direct mapped cache, all information about the cache content is lost after after a load from a single unknown address; for N-way set-associative caches, the known maximum age of all tags increases by one.

This observation leads to a promising strategy to improve the WCET performance for data caches: simply bypass the cache for accesses where the address is unknown [8]. This will never increase the WCET, as the analysis is lacking the knowledge to classify these accesses as cache hits anyway. Because the knowledge gained about the cache state is not destroyed if the access is bypassed, the precision of the analysis potentially increases.

Where as this WCET-guided optimisation is straightforward from a theoretical point of view, its implementation in a realistic setting (i.e., using an industrial WCET tool and an optimising compiler) has several demanding infrastructural requirements: (1) it requires an instruction set that allows to selectively bypass the cache when accessing the memory (2) compiler support for generating different type of memory access instructions is necessary (3) it is desirable to have value analysis results from the binary-level WCET analysis tool available in the compiler. Fortunately, these requirements are met within the context of the T-CREST project.

A peculiarity of the Patmos ISA are the typed memory instructions. The type can be one of following: *cache*, *local*, *stack*, or *bypass*. Loads typed as *cache* do access and update the data cache, fetching the contents from main memory as necessary. In contrast, loads typed as *bypass* skip the cache interaction and load the data from main memory, regardless of its availability in the data cache. Load instructions with *cache* type are potential candidates for the bypass optimisation; the compiler can emit loads of *bypass* type instead.

The set of memory accesses that should by forced to bypass the cache depends on the capabilities of the static analysis. Although it might be plausible to guess a priori which accesses will be unpredictable, it is preferable to obtain this information from the analysis itself.

In our toolchain, we read the analysis results from the XML result files produced by `aiT` and interpret the results of the value analysis in the context of LLVM's program model. The analysis tool distinguishes different contexts (e.g., call sites or loop iterations) for the address of an access (this is necessary to achieve good precision). The information for all contexts is collected separately in the `PML` database, in the same format that is used to transfer information about memory accesses from the compiler to the WCET analysis tool (see Section 2.3). As the compiler generates one block of code that is used for all contexts, eventually the results from different contexts are merged to decide whether a bypass instruction should be emitted.[2]

[2] The compiler might use procedure cloning to create multiple copies of a function for selected execution contexts. However, this optimisation might have high costs due to the increased instruction cache pressure if multiple copies of a functions are resident in the cache.

3.2.1 Implementation of Early and Late Bypass

Given the merged value range information for memory access addresses in the PML database, a memory access is classified as unpredictable if the size of the range is above a specified threshold. The corresponding memory instruction should be typed as *bypass* instead of *cache*.

We developed a compiler pass scheduled as one of the latest passes, which rewrites memory instructions to bypass the data cache, based on the memory address range information contained in a PML database. The range information is contained as *value facts* that correspond to a program point in a certain execution context. Regarding the mapping, a 1:1 mapping of basic blocks is required, i.e., only basic blocks that are progress nodes in the relation-graph model (cf. Section 2.1) between the bitcode and the machine-code representation are considered.

For each mapped basic block, a list of corresponding value facts that refer to address range information is obtained from the PML import pass. The range information is merged such that if the range exceeds a threshold (e.g., 2^{24} bytes) in *any* context, the instruction is to be rewritten. To this end, the corresponding program points are stored in a set. Once all program points for a basic block that need to be rewritten are collected, a reference to each of the program points is obtained by means of the ordinal number of the memory access, from the program representation in the PML database. Under the assumption that memory accesses are not reordered, this reference is robust even if prior optimisation passes have inserted or moved machine instructions. Therefore, in a last step the instructions of the machine basic block are traversed and load instructions are counted. Each i-th load instruction is rewritten if i is contained in the set of references.

Obviously, this approach has the disadvantage that due to loss of information in the mapping between the program representations potential candidates for bypassing the cache are missed. To overcome this limitation, we developed a tool such that the optimisation step could be delayed after code generation and linking. This *late-bypass* tool (in contrast to the bypass transformation performed early as part of the code generation) is integrated into the `platin` toolkit. Like the early bypass, the late bypass merges range information and rewrites instructions for which the range is larger as the threshold in *any* context of a program point. But, as the program representation subject to analysis and the program representation subject to optimisation are equal in this case, all instructions that are candidates for the bypass optimisation are captured.

3.2.2 Evaluation

Again, we considered the Mälardalen benchmarks and tasks from PapaBench for our evaluation. The cache configuration was same as for the evaluation of the address export (cf. Section 2.3.2), i.e., 2KB, line size 32 byte, 26 cycles transfer costs per cache line are assumed. The bypass optimisation was performed by the late-bypass tool of `platin`. Table 4 and Figure 8 depict the obtained results.

With the bypass optimisation, the WCET bound could be reduced in 50% of the benchmarks, and in 6/32 cases the WCET bound was below 90% of the original WCET bound. In three cases it even dropped down to nearly 60-65%.

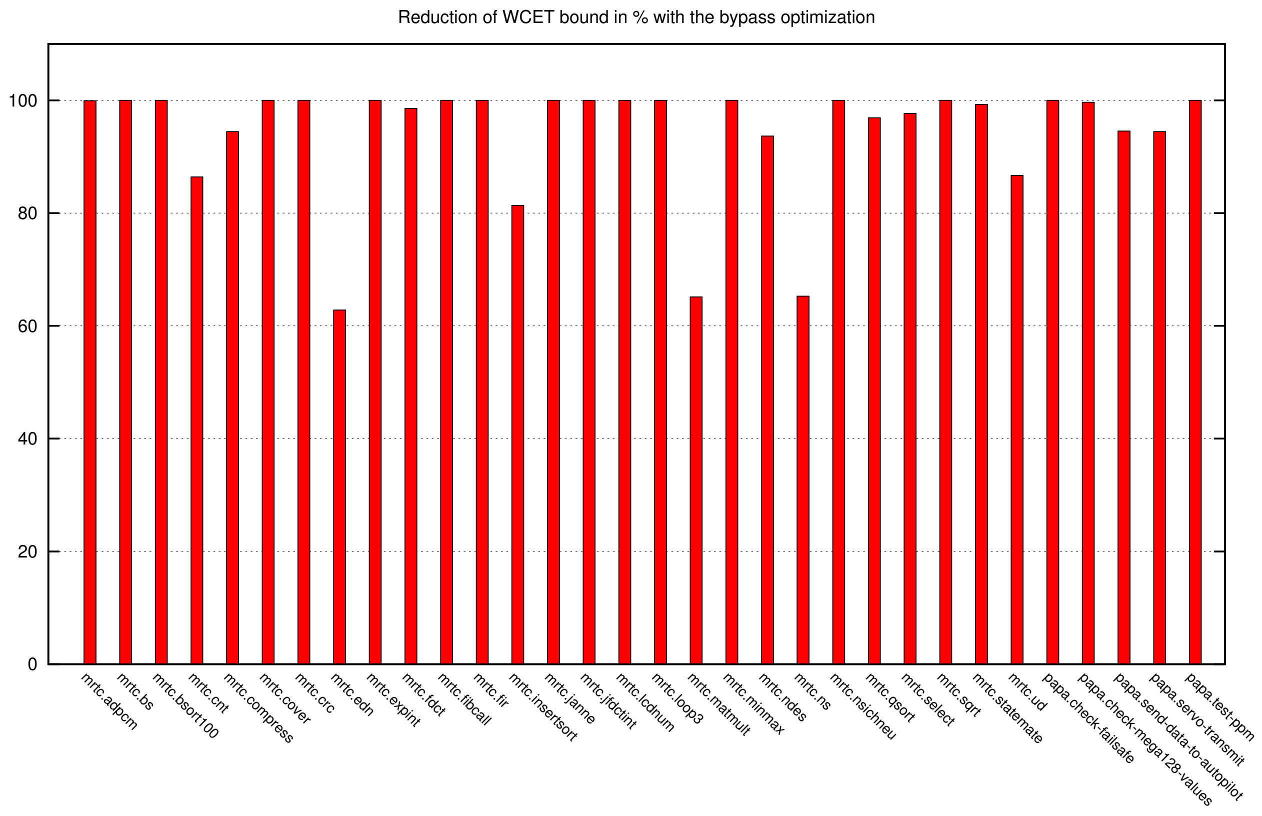


Figure 8: WCET improvement due to the cache-bypass optimisation (cf. Table 4).

benchmark	unopt	bypass	%
mrtc.adpcm	3835077	3832651	99.94
mrtc.bs	230	230	100.00
mrtc.bsort100	1131376	1131376	100.00
mrtc.cnt	11467	9909	86.41
mrtc.compress	26436	24965	94.44
mrtc.cover	1343	1343	100.00
mrtc.crc	65422	65422	100.00
mrtc.edn	390595	245364	62.82
mrtc.expint	101919	101919	100.00
mrtc.fdct	8006	7890	98.55
mrtc.fibcall	286	286	100.00
mrtc.fir	20706	20706	100.00
mrtc.insertsort	5867	4772	81.34
mrtc.janne	1060	1060	100.00
mrtc.jfdctint	10658	10658	100.00
mrtc.lcdnum	1167	1167	100.00
mrtc.loop3	9	9	100.00
mrtc.matmult	688295	448295	65.13
mrtc.minmax	524	524	100.00
mrtc.ndes	290784	272362	93.66
mrtc.ns	26985	17610	65.26
mrtc.nsichneu	16772	16772	100.00
mrtc.qsort	76210	73851	96.90
mrtc.select	35332	34505	97.66
mrtc.sqrt	9	9	100.00
mrtc.statemate	2008	1993	99.25
mrtc.ud	66043	57233	86.66
papa.check-failsafe	4243	4243	100.00
papa.check-mega128-values	4291	4276	99.65
papa.send-data-to-autopilot	2475	2340	94.55
papa.servo-transmit	2708	2558	94.46
papa.test-ppm	8731	8731	100.00

Table 4: Results from the cache bypass optimisation. Columns refer to the WCET in cycles for the program without address information, the program with address information and extended data cache analysis, and the program with bypass optimisation performed, respectively. Last column expresses the ratio $\frac{\text{WCET}_{\text{bypass}}}{\text{WCET}_{\text{no-addr}}}$.

.

3.3 Timing Analysis Feedback-driven Optimizations

Executing the aiT WCET analysis tool provides the WCET bound of the analysed task, the execution frequencies of the basic blocks on the found worst-case execution path (WCEP), as well as the WCET of all basic blocks of all analysed functions. Those results are made available to the LLVM passes via the PML import framework.

3.3.1 Deriving Criticalities for Basic Blocks

In addition to the results provided by the WCET analysis itself, the `platin` analysis driver tool calculates the criticality metric [2] for all basic blocks based on the results from aiT. The criticality of a basic block is defined as the ratio of the length of longest path over that block and the global WCET. A basic block has therefore a criticality of 1 if that block is on any worst-case path.

To derive the criticalities, we use a simplified version of the worklist algorithm for pruned criticality computation that has been presented in [1]. The `platin` driver tool iteratively runs the aiT WCET analysis while adding flow constraints that force the WCEP to cover at least one basic block that has not yet been covered. For all basic blocks v on the WCEP$_i$ found in iteration i that have not been a part of any WCEP$_j$ found in a previous iteration $j < i$, the length WCET(v) of the longest path over v is equal to the WCET found by aiT in iteration i. The criticality of a block v can therefore be calculated as WCET(v)/WCET.

This information is back-annotated to the code representation under optimisation using the generic import framework described previously. However, in general not all basic blocks of the code under optimisation have a direct mapping to analysed basic blocks. We can use a property of the criticality metric to calculate an upper bound for unmapped blocks.

Let v be a basic block of the code under optimisation that has no direct mapping to the analysed code. If a basic block w either dominates or post-dominates v, then the criticality crit(w) of w is an upper bound for the criticality crit(v) of v [2]. We therefore search for the closest dominator u of v and the closest post-dominator w of v, for which a mapping exist. The criticalities of u and w can be taken directly from the WCET analysis results. Since for the criticality of v it holds

$$\text{crit}(v) \leq \min(\text{crit}(u), \text{crit}(w))$$

we use the minimum of the found criticalities as approximation for crit(v).

The WCET analysis might calculate multiple criticalities for the same basic block, as the same basic block might be analysed in multiple execution contexts and for multiple analysis targets. The compiler on the other hand generates a single function for all contexts (see footnote 2 on page 21). To derive a single criticality value for a basic block from all execution contexts, we take the maximum of all criticalities found for that basic block.

To facilitate the implementation of criticality driven optimisations, we implemented an LLVM analysis pass that performs all the necessary steps to create a mapping of the current code representation to the analysed code, to lookup criticalities using dominators and post-dominators and to merge all results from different analysis contexts. It provides a single criticality value per basic block that can be used to guide optimisations.

3.3.2 Criticality-driven Code Optimisations

We modified the function splitter algorithm that has been presented in Deliverable D5.3 to use the criticality metric.

The function splitter creates subfunctions by iteratively adding basic blocks or strongly connected components (SCCs) to the active region until no additional basic block can be added. The original algorithm selects basic blocks based on their ID in order to preserve the original order of the basic blocks in the function where possible. The WCET-oriented function splitter instead grows regions with basic blocks that have the highest criticality. This reduces the number of branches into new subfunctions on the WCET paths, which are more expensive than branches within a subfunction. In order to avoid introducing new branches, fallthrough targets are always given a higher priority though.

Further optimisations which might profit from the criticality metric are basic block placement and if-conversion. The LLVM block placement pass rearranges the order of basic blocks so that more likely jump targets become the fallthrough target, thus eliminating the jump costs in the common case. The optimisation pass can be guided using profiling information that is collected during sample executions of the program under optimisations. We are currently extending the LLVM framework so that criticalities can be used as a static replacement for the dynamically collected profiling information. As a result, blocks with the highest criticalities will be the preferred fallthrough targets, thus minimising the jump costs on the WCEP.

On Patmos, branches within subfunctions cost three cycles, where up to two cycles can be filled with meaningful instructions. If-conversion, which eliminates branches by merging basic blocks and predicating the instructions with the branch condition, is therefore only beneficial on its own for very small basic blocks as the branch instruction is replaced by the predicated execution of *both* branches.

However, reducing the control flow results in larger scheduling regions and thus larger freedom for the instruction scheduler. The scheduler might be able to hide the costs of the if-conversion completely by scheduling instructions in the second pipeline and might even find a better schedule for the code surrounding the if-converted region. Due to hardware resource restrictions, this is not possible in all cases though.

Since the criticality $\text{crit}(v)$ of a basic block v is defined as the ratio of the longest path over v and the WCET, we can retrieve the length of the longest path over that block as

$$\text{WCET}(v) = \text{crit}(v) * \text{WCET}$$

and thus the maximum amount of cycles that we may add at block v before the WCEP switches to v as $\text{WCET} - \text{WCET}(v)$. However, since v might be executed multiple times on the longest path over v, we need to divide the result with the execution frequency $f(v)$ of v found when calculating $\text{WCET}(v)$ to get the costs for a single execution of v. The maximum amount of cycles that can be added to a single execution of v is therefore

$$\Delta c(v) = \text{WCET} \cdot \frac{(1 - \text{crit}(v))}{f(v)}$$

As a conservative heuristic, we might therefore allow if-conversion only when the code to be inserted into a block v has less cycles than this threshold. This prevents the if-converter from increasing the

WCET of a function while enabling better scheduling optimisation opportunities and a more stable execution time due to reduced control flow for non-critical code. The impact of those heuristics on real world applications remains to be evaluated though.

4 Optimisations using Patmos-specific Hardware Features

In this section we present improvements to the existing Patmos specific code generation features of the compiler that were presented in D5.3.

4.1 Instruction Scheduling for Dual-Issue Patmos

The initial support for dual-issue code generation for Patmos presented in Deliverable D5.3 consisted of several separate passes. First, a pre-register-allocation (pre-RA) scheduling pass created an initial order of the machine instructions that are generated during the code selection phase in the backend. The generic LLVM post-RA scheduler reordered instructions to minimize latencies at loads but is not able to handle bundled code. A separate packetizer pass created bundles of instructions for dual issue using basic instruction scheduling techniques. Finally, a custom delay slot filler pass reordered instructions at branches to fill branch delay slots, and inserted NOOP instructions where necessary.

The resulting quality of the generated code proved to be very low, and the individual passes were hard to maintain and to adapt to changes in the instruction set. We therefore created a new, single post-RA instruction scheduling pass, which unifies scheduling to prevent instruction hazards, creation of bundles and delay slot filling. This pass is partly based on the new scheduling framework of the latest LLVM release, which has been adopted to our needs.

The new instruction scheduler implements a bottom-up list-scheduling algorithm based on a scheduling DAG created by the LLVM scheduling framework. In the scheduling DAG, the nodes correspond to machine instructions, and edges between nodes model dependencies between instructions, such as data dependencies like read-after-write register dependencies, but also other dependencies like memory barriers or hardware resource hazards. The edges of the graph are labelled with the minimal required latency between two instructions. The latencies are derived by LLVM from an instruction itinerary table that captures the pipeline stages and the interactions between the stages of the Patmos hardware. The scheduler can therefore be much more easily adapted to changes to the hardware by simply updating the itinerary table.

Branch delay slots are modelled by artificial edges from control flow instruction to the next instruction after the branch. The latency of such an edge is set to the delay slot size of the control flow instruction, and the control flow instruction is given a high priority to ensure that it is scheduled directly before the first delay slot instruction.

The scheduler maintains a ready list of instructions of which all hazards have been resolved by the current partial schedule and thus can be scheduled at the beginning of the current schedule. If the ready list is empty, the scheduler inserts NOOP instructions as required by the ISA to prevent hazards. If multiple instructions are ready, the scheduler creates a bundle of two instructions, if possible.

If-converted code and single-path code can be scheduled efficiently by removing dependency edges in the scheduling DAG between instructions that are known to have mutually exclusive predicates and thus have no effect on each other.

4.2 Stack Cache Optimisations

In the previous deliverable D5.3 we described how the compiler supports the Patmos stack cache by moving stack objects of fixed size from the shadow stack in main memory to the specialised stack cache and by emitting stack control instructions **sres** and **sfree** to reserve and free space in the stack cache in the function prologue and epilogue, respectively.

In addition to that, the compiler must generate stack control instructions **sens** at call sites to ensure that the active stack frame resides in the stack cache, as it might have been spilled by the callee. We recall the definitions of stack occupancy and stack displacement as well as the example from Deliverable D5.3.

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.*

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.*

Example 4. *Consider a stack cache size of 4 words and a simple program with three functions* `A`*,* `B`*, and* `C` *as shown by Figure 9. The corresponding call graph, i.e., a graph representing the relation between functions calling each other, is shown in Figure 9d. 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.*

However, when we take a closer look at Example 4, 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`.

In general, it is not necessary to load back the full stack frame, or even to emit any **sens** instruction at all either if it can be shown that the stack frame is not spilled by the callee, or if not all stack slots are used between two call sites or before a return. We therefore implemented an optimisation pass that downsizes or even removes **sens** instructions where possible, which we present in the following.

4.2.1 Removing Ensure Instructions

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

```
func A() {
  sres 2;
  B();
  sens 2;
  C();
  sens 2
  sfree 2;
}
```

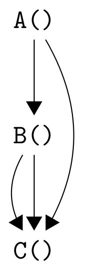
(a) Code of A

```
func B() {
  sres 3;
  C();
  sens 3
  C();
  sens 3
  sfree 3;
}
```

(b) Code of B

```
func C() {
  sres 2;
  sfree 2;
}
```

(c) Code of C



(d) Call Graph

Figure 9: A simple program consisting of three functions and their calling relations.

the ensure and its corresponding reserve operation. If the stack cache is larger than the sum of the ensure's size and the maximum displacement, the ensure can be removed without any side-effect on the program's semantics.

Computing the Maximum Displacement: The maximum displacement can be computed in two phases by (1) propagating the maximum displacement from the leaves of the call graph up towards the root and (2) by propagating information on the maximum displacement locally within functions.

In case of an acyclic call graph (without any recursion) the propagation can be done by a simple traversal of the graph in post order, i.e., the maximum displacement is computed for all children of a call graph node before it is computed for that node. This corresponds to a longest path search through a weighted call graph, where edges are assigned a weight representing the amount of space reserved in the calling function. Cyclic call graphs can then be handled by representing the longest path search through an *integer linear program* (ILP) that is solved by a standard ILP solver (e.g., CPLEX). The Patmos compiler currently uses a mixed approach that solves ILP problems for the *strongly connected components* (SCCs) of the call graph only, i.e., cyclic regions of the graph.[3] Without any additional constraints these problems are unbounded, i.e., would result in an infinitely large displacement. We thus allow the user to supply user-specified bounds on the recursion depth for individual SCCs to the compiler using a command line option (`-mpatmos-stack-cache-analysis-bounds`). We resort to a simple post order traversal for the remaining graph, where the SCCs are logically collapsed into a single node.

The second phase propagates the maximum displacement from call instructions through the control-flow graph (CFG) of individual functions to ensure instructions. This is done by iteratively propagating the maximum displacement from one node in the CFG to its successors until a fixed-point is reached, i.e., we perform a simple data-flow-like analysis. The propagation only has to consider call instructions (`call`) as well as ensure instructions (`sens`), since all other instructions cannot change the displacement. Note that the iterative processing also accounts for effects on the displacement caused by ensure operations that are about to be eliminated. Figure 10 summarises the propagation rules of this optimisation, where IN and OUT denote the maximum displacement before and after an instruction i. Figure 10a shows the propagation rule for call instructions, which states that the

[3]Formally, a strongly connected component of a directed graph $G(V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u.

$$\begin{tikzcd}[row sep=small] \text{IN: } x \arrow[d] \\ \text{i: } \texttt{call } \mathit{func} \arrow[d] \\ \text{OUT: } \max(x, \text{disp}(i)) \end{tikzcd}$$

(a) `call`

$$\begin{tikzcd}[row sep=small] \text{IN: } x \arrow[d] \\ \text{i: } \texttt{sens } y \arrow[d] \\ \text{OUT: } \min(x, |SC| - y) \end{tikzcd}$$

(b) `sens`

$$\begin{tikzcd}[row sep=small] \text{OUT: } y \arrow[dr] & & \text{OUT: } x \arrow[dl] \\ & \text{IN: } \max(x, y) \arrow[d] & \\ & \text{i: } \dots & \end{tikzcd}$$

(c) control joins

Figure 10: Propagation rules for the function-local propagation of the maximum stack cache displacement.

maximum of the initial displacement x and the displacement caused by the call instruction i (disp(i)) is propagated to the next instruction. Figure 10b shows a similar rule for ensure instructions. Since the ensure might refill the stack cache from memory, the displacement might effectively be reduced, thus the minimum of the initial displacement and the maximum displacement allowed by the ensure is propagated onward. Note that $|SC|$ denotes the size of the stack cache. At control-flow joins the maximum displacement is propagated as indicated by Figure 10c.

Example 5. *Consider again the program from Figure 9. The computation on the control-flow graph first visits the function* `C`*, which trivially gives a maximum displacement of* 2*. Next, function* `B` *is visited, which adds the displacement of* `C` *and the local displacement of* `B`*, bounded by the size of the stack cache* 4*. The displacement of* `B` *thus is* $\min(4, 3 + 2)$*.*

Using these two displacement values, we can continue with the second phase of our approach on function `A`*. The propagation starts at* `A`*'s entry with a displacement of* 0*. When the call on line 3 is reached the displacement is known to be still* 0 *before the call. The displacement of the called function* `B` *is* 4*, we thus derive that the entire content of* `A`*'s stack frame has been evicted before the ensure on line 4. The ensure thus cannot be eliminated. After the ensure* `A`*'s stack frame has been reloaded and the displacement now is* 2 *before the call to* `C`*. Since the displacement of that function is also* 2 *the displacement remains unchanged when reaching the ensure instruction on line 6. Since the value of the displacement and the argument of the ensure are equal to the stack cache size* ($2+2 = 4$)*, this ensure instruction can safely be removed.*

4.2.2 Downsizing Ensure Instructions

Apart from the elimination technique described in the previous section, we can also downsize the ensure instructions, i.e., reduce the number of ensured words, in case some data elements on the current stack frame are not live. This applies to data elements that are not used between the ensure and the next ensure, between the ensure and the next call which causes the refilled data to be spilled again, or between the ensure and a subsequent free instruction. The optimisation can also be formalised as an iterative backward data-flow problem, that propagates the size of the data area currently live in the stack cache from stack accesses (loads and or stores to the stack cache) up towards ensure instructions. The propagation rules only need to consider call instructions, stack load, stack store, as well as ensure instructions. The propagation rules are similar in structure to those in Figure 10 (we will not show them here for brevity).

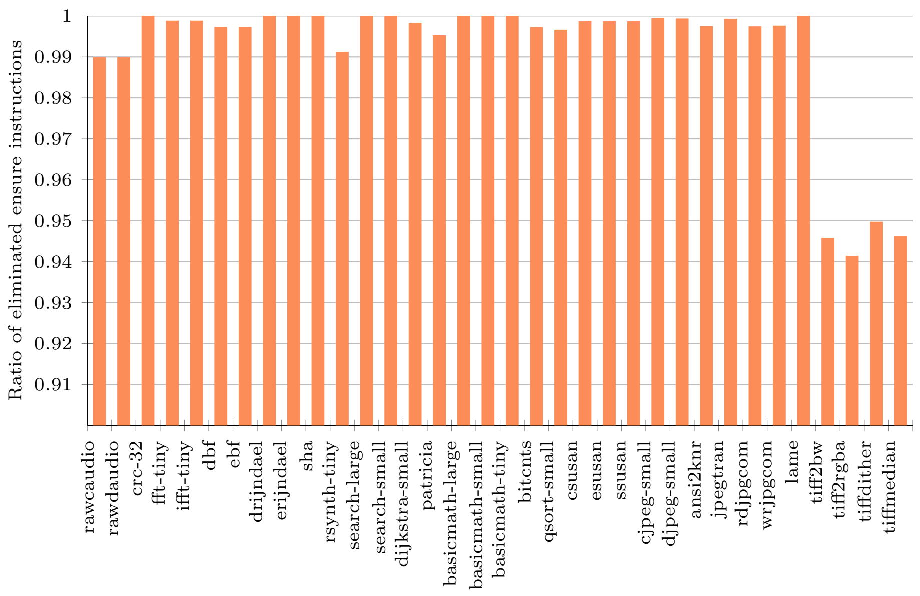


Figure 11: Ratio of ensure instructions eliminated (higher is better).

Note that the Patmos compiler performs the downsizing before the elimination of ensure instructions, as downsizing often enables the elimination of additional ensures.

4.2.3 Evaluation

We applied the elimination technique described in the previous section to a set of embedded benchmarks from the MiBench collection [4]. Table 5 shows the total number of ensure instructions in the various benchmarks, as well as the number of ensure instructions successfully eliminated (and remaining) after the analysis for a configuration of the Patmos processor with a stack cache of 2048 bytes. As can be seen almost all ensure instructions are eliminated, even in the worst-case 94% of the ensures are successfully eliminated. This indicates that our elimination strategy, even though simple, is very effective.

4.2.4 Future Directions

We are currently exploring ways to extend the notion of the two concepts of stack cache occupancy and stack cache displacement to more general models, where reserve and free operations can be placed more freely within the program's code. Well-formed programs, where each path contains well-nested sequences of reserve and free instructions having matching arguments, are promising candidates for this work. On a related note, we are exploring optimisations that allow the sinking of reserve instructions (move them down on the CFG) and the hoisting of free instructions (move them up on the CFG) within the program so as to reduce the worst-case spilling on critical paths using the criticality metric [2].

Benchmarks	Total	Remaining	Eliminated	Ratio
rawcaudio	99	1	98	0.99
rawdaudio	99	1	98	0.99
crc-32	358	0	358	1
fft-tiny	842	1	841	1
ifft-tiny	842	1	841	1
dbf	368	1	367	1
ebf	368	1	367	1
drijndael	358	0	358	1
erijndael	354	0	354	1
sha	360	0	360	1
rsynth-tiny	1,924	17	1,907	0.99
search-large	311	0	311	1
search-small	311	0	311	1
dijkstra-small	588	1	587	1
patricia	631	3	628	1
basicmath-large	839	0	839	1
basicmath-small	819	0	819	1
basicmath-tiny	826	0	826	1
bitcnts	365	1	364	1
qsort-small	590	2	588	1
csusan	760	1	759	1
esusan	760	1	759	1
ssusan	760	1	759	1
cjpeg-small	1,693	1	1,692	1
djpeg-small	1,546	1	1,545	1
ansi2knr	397	1	396	1
jpegtran	1,425	1	1,424	1
rdjpgcom	387	1	386	1
wrjpgcom	416	1	415	1
lame	3,902	0	3,902	1
tiff2bw	1,531	83	1,448	0.95
tiff2rgba	2,151	126	2,025	0.94
tiffdither	1,533	77	1,456	0.95
tiffmedian	1,486	80	1,406	0.95

Table 5: Total number of ensure instructions as well as number of eliminated and remaining ensure instructions after optimisation.

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 third year of the project. NON-CORE and FAR requirements are not listed here.

Items for which there has been progress since D5.3 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. Though, we adapted the ISA to allow storing and restoring the contents of the stack cache to and from the external memory. ***We adapted the compiler and the C library to support compilation of the RTEMS operating system, which features context switching and POSIX threads.***

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. Though, we adapted the ISA to allow storing and restoring the contents of the stack cache to and from the external memory. ***We adapted the compiler and the C library to support compilation of the RTEMS operating system, which features context switching and POSIX threads.***

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).

We adapted the `LLVM` compiler framework to automatically emit flow facts and value facts that are generated by the internal analyses performed by the compiler. The tool chain also supports generation of flow facts from execution traces. The compiler is able to emit debug information which is used by the aiT WCET analysis tool to retrieve source code level flow annotations. *Flow annotations at binary level 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. ***The compiler features WCET analysis driven optimisations and optimisations utilising the Patmos hardware features that automatically further reduce the WCET bound.***

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 and in the Patmos handbook in the patmos repository.*** *Further information about the LLVM compiler can be found in the LLVM user guide.*[4]

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. ***The compiler employs optimisation to reduce the number of emitted stack control instructions.***

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.

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 does not use the special hardware features of Patmos.

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

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.

Our tool chain extracts flow information from the code automatically and provides the information to the WCET analysis. Furthermore, because the compiler emits debug information, the capabilities of `aiT` to process annotations on the source code can be utilised.

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.

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. ***Additional value facts and flow facts are emitted by the compiler based on information generated by internal analyses of the compiler.*** *The tool chain provides means to transform this information to the input format of the WCET analysis tool.*

6 Conclusion

In this deliverable we presented improvements to the compiler and WCET analysis integration that tighten the WCET analysis results, as well as optimisations that use the information from the WCET analysis to reduce the WCET bound of tasks. We also presented an optimisation that reduces the costs of controlling the Patmos stack cache.

Since WCET analysis, compiler analyses and compiler optimisations are usually not performed on the same code representations, it is necessary to relate code representations at different optimisation levels in order to be able to transfer information between analyses and compiler passes. We implemented a framework for exporting and importing analysis results that creates a mapping between code representations without the need to adapt or disable existing compiler passes. We developed and implemented control-flow relation graphs that are used to map LLVM bitcode to the generated machine code and to transform flow constraints. By storing not only the analysis results but also the corresponding code representations as PML files, relation graphs can be constructed at any time to relate the analysis results to the code representations under optimisation.

We use this framework to export symbolic loop bounds, address information of memory accesses and stack cache occupancy information from the compiler to the WCET analysis in order to improve the precision of the WCET analysis.

In turn, the WCET analysis results are imported into the compiler using the same framework that is built around the PML file format and the Platin tool kit. This import enables leveraging the industry standard WCET analysis tool aiT as an external analysis to drive WCET-oriented code optimisations.

The compiler uses feedback from the value analysis to minimise the number of cache conflicts in the cache analysis by emitting cache bypass loads for unknown memory accesses. In our benchmarks, this optimisation is able to reduce the WCET bound by up to 60%. The WCET timing information of the WCET analysis is used to calculate criticalities, which in turn are used to drive a WCET oriented function splitter and if-converter.

Finally, we presented the instruction scheduler for the Patmos ISA, as well as a stack cache optimisation pass which is able to downsize or remove almost all stack ensure instructions in our benchmarks.

References

[1] Florian Brandner, Stefan Hepp, and Alexander Jordan. Criticality: Static profiling for real-time programs. *Real-Time Systems*. (accepted for publication).

[2] 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.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[4] 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.

[5] Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[6] Benedikt Huber, Daniel Prokesch, and Peter P. Puschner. Combined wcet analysis of bitcode and machine code using control-flow relation graphs. In Björn Franke and Jingling Xue, editors, *LCTES*, pages 163–172. ACM, 2013.

[7] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, 24(8):753–771, 2012.

[8] Thomas Lundqvist and Per Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 255–262. IEEE, 1999.

[9] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction variable analysis with delayed abstractions. In Tom Conte, Nacho Navarro, Wen-meiW. Hwu, Mateo Valero, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 3793 of *Lecture Notes in Computer Science*, pages 218–232. Springer Berlin Heidelberg, 2005.

[10] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

[11] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[12] Alexander Schrijver. *Theory of linear and integer programming*. Wiley. com, 1998.

[13] Carsten Sinz, Florian Merz, and Stephan Falke. LLBMC: A bounded model checker for LLVM's intermediate representation (competition contribution). In Cormac Flanagan and Barbara König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '12)*, pages 542–544. Springer-Verlag, 2012.

[14] Robert van Engelen. Efficient symbolic analysis for optimizing compilers. In *Compiler Construction*, pages 118–132, 2001.