T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 5.7 Report on Compiler Evaluation

Version 1.0
25 September 2014
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 José Neves Av. D. João II, Torre Fernão de Magalhães, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: jose.neves@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.	7 August 2014
0.2	Introduction and Overview.	31 August 2014
0.3	Compiler Evaluations.	10 September 2014
0.8	Pre-final version.	16 September 2014
1.0	Final version.	23 September 2014

Executive Summary

This document describes the deliverable *D5.7 Report on Compiler Evaluation* of work package 5 of the T-CREST project, due 36 months after project start as stated in the Description of Work. The deliverable comprises the evaluation of the final compiler version. In this report we first summarise the contributions of the compiler development in the scope of T-CREST, and then present the findings of the evaluation of several aspects of the the compiler tool chain such as the single path code generator and the function splitter, which have not yet been covered by previous reports. We conclude the compiler work package with an evaluation of the full compiler tool chain.

1 Introduction

The Patmos compiler tool chain is an integral part of the T-CREST platform. Based on the LLVM compiler framework [10], its main purpose is to translate any C language application to a application binary program that efficiently utilises the Patmos processor features. However, the compiler tool chain also provides means and tools to support and interact with the WCET analysis, specific optimisations targeted at the Patmos processor, as well as a single-path code generator. It is accompanied by coding policies that enable automatic detection of flow facts and more efficient single-path code transformation (Deliverable D5.5).

Compiler optimisations that target the Patmos specific stack cache and bypass memory instructions have already been presented and evaluated in Deliverable D5.4. In this report we thus focus on the function splitter and the single path code generation that have been presented in Deliverable D5.3 but not evaluated at that time.

Finally, for the full compiler evaluation, we take a look at the performance of the whole compiler tool chain, including compiler, optimisations and WCET analysis integration when applied to several typical WCET benchmarks and applications.

The rest of the document is organised as follows: First, we summarise the contributions of the compiler development in Section 2. Section 3 gives an overview of the used evaluation frameworks and discusses some of the evaluation techniques. Section 4 evaluates the single-path code generator. In Section 5, the function splitter is evaluated. Section 6 finally presents evaluations of the compiler tool chain as a whole. We discuss how the compiler meets the requirements defined in WP 1 in Section 7 and conclude in Section 8.

2 Contributions

To wrap up the research and development of the past three years, we summarise the contributions of WP5, the compiler. The compiler infrastructure itself is presented in Deliverables D5.2 (initial compiler version) and D5.6 (full compiler version).

The single contributions in the scope of the T-CREST project can be put into two categories:

1. Support for the time-predictable architecture of Patmos
2. Integration of compiler and WCET analysis

2.1 Support for the Time-Predictable Architecture of Patmos

We developed a compiler backend targeting the instruction set architecture for the time-predictable processor. The architectural features present several requirements to the compiler:

Method Cache

This cache stores large, variable-sized code segments. Loading and evicting the contents of this cache occurs only at certain instructions, rather than depending on the address fetched. This design contributes to composability, as not the code layout with the final memory addresses but the sequence of code segments fetched influences the cache state [3]. As a result, code layout optimisations in the compiler to reduce the I-cache related costs are superfluous. However, the compiler has to emit the code in chunks that fit into cache blocks.

We developed the component in the compiler backend that splits the program code into cachable segments, namely the *function splitter*. It has been first described in Deliverable D5.3. An evaluation of the function splitter can be found in Section 5, and in [6] (to appear).

Furthermore, we developed a persistence-based WCET analysis for this cache, presented in [7].

Stack Cache

The stack cache is an explicitly managed local memory for stack-allocated data (local variables, register spill locations).

We adapted the compiler to emit the required control instructions, as initially described in Deliverable D5.3. These instructions interact with the non-local memory. Optimisations to reduce the emitted control instructions have been introduced in Deliverable D5.4 and in [9].

Dual Issue

The Patmos processor features a dual-issue pipeline to provide time-predictable instruction-level parallelism. The compiler must provide a statically generated instruction schedule. The instruction scheduler that we implemented has been introduced in Deliverable D5.3.

Single-Path Code Generation

The single-path code generator targets the stability aspect of time-predictability (cf. Deliverable D5.1). The initial implementation has been described in Deliverable D5.3. We presented an improved transformation procedure in [11]. In Section 4 we present an evaluation of the implementation in the Patmos backend.

2.2 Integration of Compiler and WCET Analysis

Platin Toolkit

The `platin` tool provided by the Patmos tool chain builds the bridge between the compiler and the WCET analysis. It relates program annotation information emitted by the compiler at different compilation phases and code representations with the final application binary under analysis. The

`platin` tool also transforms and simplifies flow facts, provides analyses for Patmos caches, generates input files and program configurations for the compiler, the simulator and the WCET analysis tools, and provides an alternative, simplified front-end to the compiler tool chain. Basic support for WCET analysis within the compiler tool chain using `platin` has been introduced in Deliverable D5.3, while techniques to improve the results of the WCET analysis and to enable interaction between the compiler and the WCET analysis have been detailed in Deliverable D5.4 and in [12].

Loop-Bound Analysis

We utilised an analysis pass of the compiler infrastructure to process and export the information available about the evolution of scalar variables during loop iterations. Our tool chain infers symbolic and constant loop bounds from the knowledge about maximum header trip counts, and eventually publicises this information to the timing analysis tool. This partly saves the programmer from the tedious (and error-prone) task of manually annotating loop bounds. This aspect of compiler–analysis integration has been presented in Deliverable D5.4.

Address Export

Similar to knowledge about loop bounds, information about target addresses of memory access instructions available in the compiler is processed and exported to the timing analysis tool by our tool chain. Our implementation has been described in Deliverable D5.4.

Bypass Optimisation

Data accesses to unknown memory locations degrade WCET-analysability, as for cache analysis it must be assumed that accesses potentially target any cache line. The WCET analysis tool reports such memory accesses classified as “imprecise”. We implemented an optimisation, that - based on the information of the timing analysis tool - bypasses the data cache for such imprecise memory accesses, resulting in improved data cache analysis results. This optimisation has been introduced and evaluated in Deliverable D5.4.

Criticality Metric

We developed a metric obtained from the WCET analysis results, that describes the “criticality” of a basic block w.r.t. the worst case path. This metric has been published in [1, 2]. In Deliverable D5.4 we present how this criticality metric can be used to drive decisions in the function splitter.

User-supplied Source Code Annotations

In the scope of the EU COST action TACLe, efforts are made to establish a representative set of timing analysis benchmarks in the WCET community. The source level annotations have a defined format. In the last project phase, we implemented support for the TACLe loop-bound `#pragma`

annotations in the compiler tool chain. This new feature allows compiler users to provide and utilise user-supplied annotations within the compiler, including their co-transformation during optimisation passes, and export to the timing analysis tool. We briefly describe these annotations in Section 4.

Furthermore, our tool chain supports a couple of platin-specific annotations, which are not described in detail here.

3 Methodology

For our evaluation, we use the Mälardalen [4] and the MiBench [5] benchmark suites as well as CoreMark.[1] All benchmarks were compiled with the Patmos tool chain, using the default compiler optimisation level -O2 if not otherwise noted. The default hardware configuration that is used is a dual-issue Patmos core using a 4 kB variable sized method cache, a 2 kB direct mapped data cache and a 2 kB stack cache if not otherwise noted. Runtime evaluations were preformed using the Patmos simulator, while the AbsInt a3 tool was used to perform WCET analyses.

3.1 Flow-Facts

In order to perform WCET analyses, flow facts are required that bound the number of iterations of all loops. Sets of possible jump targets of indirect branches and indirect calls must be determined. In some cases such flow facts can be derived automatically by the WCET analysis tool. The compiler supports the analysis by providing flow facts already known to the compiler and also performs some flow analyses on its own. However, if no bounds or only very imprecise bounds can be derived, manual annotation is required.

In order to avoid tedious[2] and error-prone[3] manual annotations, we use the simulator as an additional source for flow facts. `platin` is able to analyse simulation traces and derive *trace* facts with arbitrary precision from the traces (local or global bounds, context-sensitive or context-insensitive bounds, various types of trace facts). Trace facts are not safe in the general case with arbitrary inputs, but hold for a limited, known set of inputs, as it is the case for benchmarks.

The trace facts are derived on the binary under analysis, complex and potentially lossy flow fact transformations are not required. Since the trace facts are derived from the actual execution of the program, the trace facts constitute the most precise flow facts possible. This has the advantage that it reduces analysis imprecision due to imprecise flow facts. However, the flow facts derived from the trace analysis still over-approximate the actually executed path, since trace facts emitted to the WCET analysis use limited or no context-sensitivity, i.e., flow facts for different execution contexts of a loop

[1] `http://www.eembc.org/coremark/`

[2] Library functions from the C library are not written or documented with analysis in mind and are often very hard to impossible to manually analyse.

[3] A common source for erroneous annotations proved to be the difference between loop bounds and loop header trip counts. The latter are typically used by analysis tools, while programmers tend to think in loop bounds, which do not include the exit check of the loop. This can often lead to underapproximations of the loop bounds by one iteration. Static analysis as performed by AbsInt a3 can detect such annotation errors in some but not all cases, leading to hard to detect analysis errors.

or instruction are merged into a single flow fact. The default setting for the trace-analysis generates context-insensitive loop bounds, infeasibility information and indirect call target information per analysis target function entry. The trace analysis generates flow facts for all executed code. Code that is not executed is marked as infeasible. As a result, manual annotation is still needed for code that is not executed in benchmarks for analyses that do not use infeasibility information, e.g., the `platin` cache analyses. `platin` provides a tool that allows users to add and modify such flow annotations in the PML annotation files.

3.2 Source Level Flow Fact Annotations

While support for source level flow annotations is not a central aspect of the T-CREST project, some initial support for source level annotations has been added to `patmos-clang` and `platin`, because they help to obtain tighter WCET bounds. The annotations are based on unique *markers*, which are placed by the programmer into the code. The markers are assumed to have side-effects by the compiler. The compiler is thus not allowed to reorder or remove markers on any path. If code is duplicated, markers are duplicated as well. On any possible execution path of the program, the sequence of markers on that path is invariant for any code transformation pass. While this can impose some restrictions on certain optimisation passes, it allows the programmer to specify flow facts in terms of markers.

Marker-based source-level flow annotations proved to be quite powerful for bounding loops and control flow. However, markers are not sufficient to annotate indirect call targets or value facts such as function argument values. In future work, one possibility to add support for such annotations could be to define markers on variables by passing variables as arguments to the markers. Such markers can then be used to specify value domains for variables at certain program points. However, this would require the extension of `patmos-clang` and `platin` to provide a framework to analyses to relate such value markers and the variables of interest in the code.

In most of our evaluations, we only use trace facts found by the simulator and flow facts found by the compiler or the WCET analysis. For the single-path code evaluations, we also use code level loop bound annotations that assist the single-path code transformation. The loop bound annotations provide a simple way to define context-insensitive local loop bounds, which are the most common case of flow annotations and are required by the single-path loop transformation.

4 Single-Path Code Generation Evaluation

The Patmos compiler features support for single-path code generation. The single-path code generator is a set of compilation passes that transforms code to machine code with a singleton execution trace. A first version of the implementation was presented in Deliverable D5.3. Since then, it has been extended to support a more general class of input programs [11], and by support for inter-procedural transformation. Also, the usage of loop bound annotations, which was not present in M18, was added in the final version.

The timing of a piece of code is determined by two aspects: First, the sequence of actions along an execution path, and second, the durations of these actions. The single-path approach addresses

the first aspect to obtain execution time behaviour with low jitter (*timing stability*), i.e., get equal execution times for varying input data. The second aspect is dependent on the underlying hardware architecture. If the duration of actions can be made constant w.r.t. the execution context, constant execution times can be achieved in conjunction with the single-path approach.

Hence, we used following different benchmarking objectives:

1. Validate that the generated single-path code has a singleton execution trace.
2. Investigate the influence of the memory hierarchy on the execution time of the singleton execution path.
3. Evaluate two optimisations implemented in the single-path code emitter.

4.1 Singleton Execution Paths

To enable the single-path conversion in the compiler, the functions have to be specified for which single-path code should be generated. These functions typically are called within a hard-real time task, e.g. to process some input or to handle an external event. Functions called within these entry functions are duplicated and one copy is converted to a single-path variant. This immediately leads to a limitation of the single-path code generator: Call targets must not be computed dynamically (indirect calls), but must be known statically.

As for all timing-analysable functions, loop bounds must be known beforehand at compilation time. To enable the specification of local loop bounds that are usable by the compiler – as opposed to annotations solely for analysis purposes by the timing analysis tool – we added support for the loop bound pragmas that are used in the TACLeBench benchmarks. The have to lexically precede `while`-, `for`- and `do`-statements and have the following syntax:

```
#pragma loopbound min <NUM> max <NUM>
```

The values for `NUM` have to be constants. In the single-path code generator, the maximum value is used for generating input-data independent loop iteration counts.

The benchmarking objectives require that the benchmarks contain target functions that accept different inputs, which cause different execution paths to be taken. To this end, we adopted a subset of the Mälardalen benchmarks and created multi-path variants that can be executed.

To benchmark the generality of the transformation, we compared the simulation for the conventional variant with the single-path variant. We assumed an ideal memory, with single-cycle memory access latency. We included the statically obtained execution time bounds for the conventional variant.

Results

Table 1 depicts the results. The functions are labeled as “benchmark name”/“entry function”. For each function the execution times are given as interval or as a single constant if all observed execution

Function	Conventional	Static Analysis	Single-Path
bs/binary_search	[79, 114]	114	129
bsort100/BubbleSort	[60907, 70403]	139914	178222
cnt/Test	[3602, 3616]	3657	3908
compress/compress	[3443, 3687]	N/A	[45059, 49261]
cover/swi10	[131, 131]	131	1258
crc/icrc	[625, 30071]	30101	39232
expint/expint	[488, 82215]	95159	236380
fac/fac_sum	[171, 287]	N/A	[216, 353]
fdct/fdct	[1447, 1447]	1447	1499

Table 1: Simulation results in an ideal (no memory access latencies) setting, for the multi-path Malardalen benchmarks.

times were equal. In this evaluation, only violations of the single-path property lead to varying execution times in the single-path variant. Note that in the conventional variant the inputs may not lead to the longest program execution path.

We give some remarks on the benchmarks to discuss these figures, and give deeper insights into the capabilities and limitations of the single-path code generator. They also give some hints about the effect of a particular coding style on the performance of the generated single-path code. Hence, this evaluation is meant to be of qualitative nature.

bs/binary_search
: This small function has a slightly longer single-path execution than the maximal observed conventional execution.

bsort100/BubbleSort
: The bubble sort function contains a so-called triangular loop, where the inner iteration count is dependent on the outer iteration counter such that for each iteration of the outer loop, the inner loop executes as many times as the value of the outer iteration counter. This triangular loop is transformed to a rectangular loop, such that the inner loop always executes the local maximum number of iterations, as specified by the loop bound annotations.

cnt/Test
: In this benchmark, the single-path variant actually has a shorter execution time than any of the executions of the conventional variant. This is due to the tight code the scheduler is able to produce with the if-converted loop body.

compress/compress
: The source code of the compress function contains goto statements that form irregular loop constructs. With the current loop bound annotations it is not possible to specify bounds for these irregular structures. We did not provide any additional annotations, therefore we did not obtain any results by static analysis. Furthermore, the loop body contains calls to costly functions that are only called in a few iterations in the conventional version, but in every iteration in the single-path version.

cover/swi10
: The benchmarked function consists of a large switch statement inside a loop (iterating 10 times), where each case is handled exactly once, i.e., in each iteration, another case block is executed. The single-path code generator contains a pass that transforms a `switch`-statement into cascade of `if-then-else` statements. In the resulting execution path, *every* case is visited in each iteration, such that large switch statements contribute to a high execution cost in the singleton execution path. We observed an almost 10-fold increase in the execution time.

crc/icrc
: The icrc function executes a costly initialisation function and sets a global flag. On successive executions, the flag is checked and the initialisation code is skipped and an alternative code path is executed. The single-path variant fetches both pieces of code in every execution.

expint/expint
: A large `if-then-else` construct dominates the expint function. Each of these alternatives contains loops with a division in the loop body. Division is implemented as software library call and expensive in terms of execution cycles. Because the call to the division function is fetched and performed twice as many times in the worst case in the single-path variant than in the conventional variant, the execution path length is about twice as high.

fac/fac_sum
: The benchmark contains a call to a recursive function to compute the factorial of a number. In general, recursive functions are not viable for single-path transformation. Interestingly, LLVM eliminates the recursion, but no loop bound annotations can be applied for the newly created loops in the single-path conversion, leading to varying execution times. Due to the lack of additional annotations, also no static analysis result is obtained.

fdct/fdct
: Although the code does not contain alternatives and variable iteration counts, the produced code is longer as the single-path generator is reserving registers for the predication pass, which leads to a higher register pressure during the earlier register allocation and becomes visible in this computationally intensive benchmark.

The results have shown a drastic increase in execution time in some cases, due to the serialisation of control-flow alternatives. However, the benchmark code did not require any modifications to enable the single-path transformation (apart from loop bound annotations), showing that automated conversion to single-path code within a compiler backend is feasible. In this regard, this work is extending the previous state-of-the-art in the field.

4.2 Influence of the Memory Hierarchy

As mentioned in the introduction, the timing of a single-path execution is dependent on the duration of the operations along the singleton execution path. The timing of memory operations – given their predicate is true – is dependent on the internal state of the particular memory component.

Patmos has different types of local memories: the method cache, the stack cache, the data cache, and the data scratchpad. This architecture simplifies WCET analysis of single-path code due to its

Function	Conventional	Static Analysis	Single-Path
bs/binary_search	[157, 202]	218.0	[219, 229]
bsort100/BubbleSort	[62195, 119211]	377372.0	[179530, 227050]
cnt/Test	[4483, 4497]	4554.0	4815
compress/compress	[8769, 9103]	N/A	[50334, 55481]
cover/swi10	[190, 190]	219	1405
crc/icrc	[655, 31626]	27044.0	[39232, 40808]
expint/expint	[677, 82414]	95287.0	236649
fac/fac_sum	[202, 318]	N/A	[251, 388]
fdct/fdct	[2337, 2337]	3223.0	2401

Table 2: Simulation results with the full configuration for memory access latencies, for the multi-path Malardalen benchmarks.

composable nature. Because the sequences of (sub-)function calls and stack cache control operations are invariant for single-path executions, the method- and stack-cache related memory access worst-case costs can be easily obtained by a simple simulation.[4] These memory access related worst-case costs can be added to the costs of the singleton execution path in the ideal memory case.

Accesses to static and heap data make use of the data cache. They are predicated and as such their effect and latency is dependent of the predicate value. However, as the *potential* memory access sequences in the single-path variant are not different from the conventional variant, one could add the data-cache related memory access cost obtained from static analysis of the conventional variant.

We investigated the execution times in the presence of caches and memory access latencies. We used a cache configuration labelled *full*, where we assumed a burst access time of 5 cycles and a single beat access latency of 2 cycles. In addition, we used a configuration labelled *data0*, where we assumed an ideal data cache where every memory access is a cache hit with no additional access latency.

Results

The results are depicted in Table 2. The single-path variants of the benchmarks bs/binary_search, bsort100/BubbleSort, and crc/icrc expose a variable execution time behaviour with this configuration, in addition to the benchmarks in Table 1. The execution time variance stems from the memory accesses through the data cache. When assuming no access latencies for data accesses through the data cache (always hit), the execution times are invariant, as shown in Table 3.

4.2.1 EEMBC AutoBench Benchmark Results

To benchmark a broader range of applications, we also adapted a small number of the EEMBC AutoBench Benchmark suite. Modifications were required to factor the critical code out to separate functions. The benchmarks contain test data as inputs, which trigger different execution paths in these critical functions. Again, we used the three different configurations: *ideal*, *data0* and *full*.

[4] In the presence of constant memory transfer times between the main and the local memories, the worst case costs would be equal to the actual (invariant) execution costs.

Function	Conventional	Static Analysis	Single-Path
bs/binary_search	[122, 157]	157	184
bsort100/BubbleSort	[60950, 70446]	139957	178285
crc/icrc*	[655, 30156]	30186	[39232, 39338]
crc/main_test	[6842, 36319]	65795	84589

Table 3: Simulation results with the data0 configuration, i.e., the same configuration as in Table 2 but an ideal data cache. (*) Function icrc is called twice in the main_test driver function, which leads to differing cache states w.r.t. the method cache for these invocations.

Configuration	Benchmark	Conventional	Static Analysis	Single-Path
ideal	a2time	[587, 622]	1547	1241
	bitmnp	[16122, 16835]	20373	28015
	tblook	[2411, 3902]	12442	12643
data0	a2time	[1396, 1484]	2769	2621
	bitmnp	[21693, 22804]	22902	28444
	tblook	[7795, 11328]	19869	23516
full	a2time	[1451, 1544]	2918	[2666, 2681]
	bitmnp	[26783, 30684]	36237	[38389, 38894]
	tblook	[7850, 11383]	20929	[23551, 23806]

Table 4: Results for the evaluated EEMBC AutoBench benchmarks.

The results are depicted in Table 4.

The single-path variant of the a2time benchmark outperforms the statically determined bound in all configurations. This can be attributed to the fact that the analysis tool is conservative about alternatives that are mutually exclusive by program semantics. Furthermore, the scheduler can generate more compact code for the predicated instructions in the single-path variant, than for the branching code in the conventional variant.

The bitmnp benchmark code contains a nested if-statement with large, balanced alternatives. As the single-path variant fetches all the alternatives, this leads to a high increase in execution time, and a large gap between the static analysis bound and the single-path simulation in the ideal and data0 configurations. With access latencies to data, the gap is smaller.

The code for the tblook benchmark uses software floating point routines. As a result, the single-path execution cost is almost twice as high for the data0 configuration than for the ideal configuration. In the latter, the execution time of the single-path variant is only 1.6% higher than the static analysis bound. The gap is higher in the other configurations as the amount of code required to be loaded to the method cache is large, leading to a higher cumulative access latency.

4.3 Single-Path Code Emission Optimisations

We developed two optimisations for the single-path code generator, during the code emission phase:

- In the Patmos ISA revision (starting M23), a new instruction was introduced, namely, the `bcopy` instruction. This instruction copies the value of a predicate to a specified bit position of a general purpose register. With this instruction, predicate spilling and restoring should become more efficient, as opposed to working with bitmasks. We call this optimisation *bcopy* optimisation.
- The fact that Patmos allows to copy the predicate register file as a whole led to the development of the predicate register allocator that is loop-scope based. Upon entry the predicate register file is stored, a new set of predicate is allocated, and upon exit the former predicate register file restored. The optimisation avoids unnecessary spills (and restores) and has already been described in D5.3. In the following, we call this optimisation *nospill* optimisation.

For the evaluation, we used the set of standard Mälardalen benchmarks, with no modifications or loop-bound annotations.[5] We conducted the experiments without any of the two optimisations, with each in separately, and with both enabled.

Table 5 shows some statistics for the generated single-path code with the optimisations disabled, which is the baseline for the following experiments. Two benchmarks (nsischneu, ns) do not compile at all because of the current limitations of the single-path code generator.[6] The second column depicts the number of required predicates (sum over all functions). The third column shows the number of branch instructions removed by the conversion. The forth column shows the number of inserted instructions. The rightmost column shows the simulated execution time. If no number is shown, then the benchmark was either optimised out (fft1, fac), or it did not terminate (also not in the conventional variant).

Table 6 shows the results for of the bcopy optimisation. The second column shows the execution time, and the third column the speedup in percent of the execution time. The fourth column shows the number of inserted instructions, and the fifth column shows the reduction of the number of inserted instructions in percent. The improvements are mostly below 1%, both w.r.t. speedup and number of inserted instructions. For smaller benchmarks (bs, cover, lcdnum, prime), the benefit seems to be higher (up to 12%).

Table 7 shows the results for the nospill optimisation. The second column shows the number of loop scopes, where spilling and restoring of the predicate register file could be omitted. The other columns describe the same metrics as before. The payoff of this optimisation seems to be higher, i.e., up to 22% speedup for smaller benchmarks and up to 52% fewer instructions inserted.

The results with both optimisations enabled are shown in Table 8. The improvements are dominated by the nospill optimisation.

4.4 Discussion

We have evaluated different aspects of the single-path code generator.

[5] The backward branch condition in a loop is based on the value of the header predicate in this case.

[6] In the current implementation, the number of predicates in a function is limited to 64. Also, exit edges of nested inner loops must not exit the outer loops as well.

Benchmark	#Pred	#BrRm	#Insrt	ET
adpcm	36	31	133	1160
bs	4	3	17	110
bsort100	6	6	17	85850
cnt	7	6	28	3678
compress	33	28	134	44443
cover	7	6	20	248
crc	7	6	22	31651
duff	3	2	8	192
edn	34	33	122	76971
expint	9	6	32	130148
fac	0	0	0	-
fdct	3	2	14	1499
fft1	0	0	0	-
fibcall	2	1	7	277
fir	6	5	17	3208
insertsort	6	5	17	697
janne_complex	6	5	18	405
jfdctint	3	2	14	1658
lcdnum	45	44	107	161
lms	313	275	940	-
loop3	1	0	0	10
ludcmp	147	131	458	151783
matmult	8	7	45	155007
minmax	8	6	16	125
minver	163	143	477	50705
ndes	34	34	106	40552
prime	11	8	54	270080
qsort-exam	42	37	114	7127
qurt	166	144	507	117649
recursion	6	4	21	-
select	36	31	78	-
sqrt	0	0	0	-
st	166	143	408	1746050
statemate	25	23	79	358
ud	17	16	87	16076
whet	482	418	2485	430943608

Table 5: Single-path statistics for the standard Mälardalen benchmarks.

Benchmark	ET	Speedup in %	#Insrt	Red. No. %
adpcm	1160	0	133	0
bs	102	7.27	15	11.76
bsort100	85850	0	17	0
cnt	3678	0	28	0
compress	43904	1.21	129	3.73
cover	228	8.06	18	10
crc	31651	0	22	0
duff	192	0	8	0
edn	76971	0	122	0
expint	130148	0	32	0
fac	-	N/A	0	N/A
fdct	1499	0	14	0
fft1	-	N/A	0	N/A
fibcall	277	0	7	0
fir	3208	0	17	0
insertsort	697	0	17	0
janne_complex	405	0	18	0
jfdctint	1658	0	14	0
lcdnum	146	9.32	107	0
lms	-	N/A	939	0.11
loop3	10	0	0	N/A
ludcmp	151022	0.50	454	0.87
matmult	155007	0	45	0
minmax	125	0	16	0
minver	50426	0.55	473	0.84
ndes	40552	0	106	0
prime	268506	0.58	48	11.11
qsort-exam	7063	0.90	112	1.75
qurt	116465	1.01	505	0.39
recursion	-	N/A	21	0
select	-	N/A	78	0
sqrt	-	N/A	0	N/A
st	1746050	0	408	0
statemate	344	3.91	79	0
ud	16076	0	87	0
whet	430939128	0.00	2480	0.20

Table 6: Single-path statistics with the `bcopy` optimisation enabled.

Benchmark	# of nospills	ET	Speedup in %	#Insrt	Red. No. %
adpcm	17	1126	2.93	69	48.12
bs	1	86	21.82	8	52.94
bsort100	1	85454	0.46	14	17.65
cnt	2	3598	2.18	22	21.43
compress	3	44239	0.46	123	8.21
cover	1	193	22.18	11	45
crc	3	30619	3.26	13	40.91
duff	0	192	0	8	0
edn	15	76452	0.67	66	45.90
expint	1	129348	0.61	29	9.38
fac	0	-	N/A	0	N/A
fdct	0	1499	0	14	0
fft1	0	-	N/A	0	N/A
fibcall	0	277	0	7	0
fir	1	3201	0.22	13	23.53
insertsort	2	647	7.17	9	47.06
janne_complex	1	360	11.11	14	22.22
jfdctint	0	1658	0	14	0
lcdnum	0	161	0	107	0
lms	3	-	N/A	931	0.96
loop3	0	10	0	0	N/A
ludcmp	7	151560	0.15	434	5.24
matmult	4	153167	1.19	33	26.67
minmax	0	125	0	16	0
minver	8	50573	0.26	448	6.08
ndes	9	40279	0.67	74	30.19
prime	3	264416	2.10	31	42.59
qsort-exam	2	6817	4.35	101	11.40
qurt	0	117649	0	507	0
recursion	2	-	N/A	14	33.33
select	1	-	N/A	74	5.13
sqrt	0	-	N/A	0	N/A
st	4	1746050	0	392	3.92
statemate	0	358	0	79	0
ud	7	15762	1.95	64	26.44
whet	22	430569308	0.09	2384	4.06

Table 7: Single-path statistics with the `nospill` optimisation enabled.

Benchmark	ET	Speedup in %	#Insrt	Red.No. %
adpcm	1126	2.93	69	48.12
bs	86	21.82	8	52.94
bsort100	85454	0.46	14	17.65
cnt	3598	2.18	22	21.43
compress	43700	1.67	118	11.94
cover	193	22.18	11	45
crc	30619	3.26	13	40.91
duff	192	0	8	0
edn	76452	0.67	66	45.90
expint	129348	0.61	29	9.38
fac	-	N/A	0	N/A
fdct	1499	0	14	0
fft1	-	N/A	0	N/A
fibcall	277	0	7	0
fir	3201	0.22	13	23.53
insertsort	647	7.17	9	47.06
janne_complex	360	11.11	14	22.22
jfdctint	1658	0	14	0
lcdnum	146	9.32	107	0
lms	-	N/A	930	1.06
loop3	10	0	0	N/A
ludcmp	150794	0.65	430	6.11
matmult	153167	1.19	33	26.67
minmax	125	0	16	0
minver	50294	0.81	444	6.92
ndes	40279	0.67	74	30.19
prime	264416	2.10	31	42.59
qsort-exam	6803	4.55	101	11.40
qurt	116465	1.01	505	0.39
recursion	-	N/A	14	33.33
select	-	N/A	74	5.13
sqrt	-	N/A	0	N/A
st	1746050	0	392	3.92
statemate	344	3.91	79	0
ud	15762	1.95	64	26.44
whet	430572508	0.09	2383	4.10

Table 8: Single-path statistics with both optimisations enabled.

The implementation is capable of automated conversion to single-path code of functions that are bounded by local loop bounds.

However, in some cases, a major increase in execution time was observed. To overcome the degradation in execution time, one possibility would be – especially when writing new code artefacts – to adhere to a WCET-oriented coding style i.e., to avoid input-data dependent control-flow in the first place. This includes hard to analyse and hard to annotate constructs like special case treatment in the first or last loop iteration (e.g. initialisation, or unstructured code like loops that cannot be easily annotated).

Another possibility would be to develop compiler optimisations tailored to single-path code generation. For example, a transformation performing the inverse of loop unswitching could allow the single-path backend to produce more compact code. We consider this subject to future research.

Furthermore, one could relax the property of a singleton execution path and make the conversion *mode-specific*. For example, in code with large alternatives, the condition of this alternatives could be treated as program mode. As a result, the alternatives would not be serialised but rather expose two different paths depending on this (explicit) program mode. The execution time could be reflected in a simple formula parametric on the program modes. However, the number of modes should be kept small. In this regard, the evaluation provided valuable insights and directions for future research.

Patmos provides a data scratchpad memory which can be utilised to avoid the execution time variability resulting from data accesses through the data cache. In a usage scenario, one would copy the data the task is going to operate on from the global memory to the local scratchpad memory. Hence, no interactions with the global memory would occur during task execution. After finishing computation, the data is copied back to the global memory. This adheres the S-Task model, which was discussed in deliverable D5.1 (Section 3.2). Following this model, it is assumed that each input data is available locally before the task starts. Also, data is not written back to global memory before the task has finished. This way, the computational part does not interfere with global memory access, and the concerns of computation and communication are separated.

While adhering to the S-Task model might require substantial redesign of existing legacy applications, it has the benefit that one could directly communicate with other Patmos processor cores without passing data through the global memory.

5 Function Splitter Evaluation

The compiler supports the Patmos method cache by splitting functions into smaller blocks of code that are guaranteed to fit into the method cache. The processor thus does not need to implement any – potentially complex – mechanisms to deal with functions that are too large to fit into the cache as a whole. Furthermore, splitting functions into smaller code blocks reduces the chance of smaller functions to be unnecessarily evicted from the cache by large blocks of (temporarily) unused code. The function splitter also enables the use of a fixed-block method cache implementation, where the maximum code-block size is further limited by the hardware, but allowing for an LRU replacement policy and a simpler integration of the cache analysis into traditional data-flow based analysis frameworks, as employed for example by the AbsInt a3 WCET analysis tool.

Emitting each basic block as a separate code block would provide a valid, trivial splitting. However, the overhead caused by necessary code inserted at code-block transitions and the increased number of conflict misses due to the limited tag memory size make it necessary to emit larger code blocks. We presented an initial function splitter algorithm in Deliverable D5.3 that forms single-entry regions on the control-flow graph to emit code blocks of approximately similar size based on a given size limit. More details on the algorithm can be found in [6]. The function splitter works on the basic principle of growing regions up to a given size threshold on an acyclic control-flow graph (CFG). Regions are grown by adding basic blocks to regions while maintaining the single-entry property of the regions. Cyclic and non-natural CFG are handled by transforming the graph into an acyclic graph while maintaining relevant dominance properties.

In this report, we focus on the evaluation of the implemented function splitter. For this evaluation we used the Mälardalen, MiBench [5] and CoreMark benchmark suites. We used a single-core dual-issue Patmos configuration in which memory is accessed in bursts of 4 words taking 7 cycles per burst. By default, we used a FIFO method cache of 4 kB that can hold up to 32 variable sized code blocks. The implementation of the variable sized method cache used in Patmos is described in more detail in [3]. The method cache is configured to use up to 1 kB large page bursts, i.e., after an initial 7 cycles for the first 4 words, every additional word costs only one cycle.

5.1 Compiler Evaluation

First, we take a look at the performance of the function splitter itself. Table 9 gives the number of functions splitted, the number and average size of regions emitted and the run time of the function splitter pass in the `patmos-clang` compiler when applied to all individual benchmarks. The table gives the results for various maximum code block size grow limits. Note that regions can be significantly smaller than the specified region size limit as a region includes at most a single function and many functions are much smaller than 1 kB. Regions can also be larger than the given limit (up to the size of the method cache), since a single large basic block is only split if it is larger than 2 kB of code.

The number of functions that need to be split and the number of regions drops when larger regions are emitted. When regions are only grown up to a size of 32 bytes, nearly all basic blocks become regions on their own. In this setting, the code size increase of the function due to jump instructions that need to be inserted at fall-throughs is significant in relation to the average region size. This is

Region grow limit (bytes)	32	64	128	256	512	1024
Functions splitted	8685	8579	7175	4500	2869	1434
Regions emitted	196972	182122	123551	73927	43229	25158
Regions per function (avg/max)	22.0 / 1377	21.0 / 1302	17.0 / 868	16.0 / 539	15.0 / 331	17.0 / 215
Basic blocks per region (avg/max)	1.0 / 13	1.1 / 13	1.6 / 15	2.5 / 15	3.9 / 18	5.5 / 31
Region size (avg/max) (bytes)	49.9 / 2040	53.0 / 2040	71.0 / 2040	104.6 / 2040	158.6 / 2040	224.9 / 2040
Code size increase (avg/max) (bytes)	12.3 / 17436	12.4 / 16328	12.2 / 10520	13.2 / 8292	15.5 / 6612	19.2 / 5772
Total code size increase (kB)	2372.3	2199.8	1473.0	955.0	652.8	470.7
Max run time per function (ms)	40.9	28.7	27.8	25.1	25.8	25.7
Total run time (ms)	1358.9	1366.0	1244.4	1130.0	1033.4	936.6

Table 9: Emitted regions and run time of the function splitter pass in `patmos-clang` over all benchmarks.

reduced by allowing for larger regions in the settings in the right-most columns of the table. Splitting all functions takes up less than two seconds of the whole compile time of 225 seconds to compile all benchmarks on an Intel Core i5-2520M CPU at 2.50GHz with 8 GB of RAM, using a release build of `patmos-clang`.

5.2 Runtime Evaluation

To evaluate the impact of the function splitter on the runtime of the benchmarks, we selected 9 benchmarks from the Mälardalen suite and 20 benchmarks from MiBench that execute at least 4 kB of code. Each benchmarks executes between 6 to 93 different functions with a total of 4 to 117 kB of code.

We evaluated the function splitter using different code-size limits for growing the regions. Since loops might be expected to reuse larger amounts of their loop body than acyclic regions with conditional control flow, we also explored different limits for (reducible) loops that do contain calls and for acyclic code regions. The function splitter setups in this section are thus represented as *two values*, the first value being the region grow limit for acyclic code, the second value being the limit for emitting a loop body as a single region. Independent of the aforementioned region grow limits, the function splitter always tries to add a block to a region that has no calls when the block post-dominates the entry block. Such a block will always be needed in the cache when any part of the region is executed. Similarly, basic blocks are only split into multiple code blocks if they would not fit into the whole method cache otherwise.

The performance of an application heavily depends on the actual size of the code blocks. Small blocks incur large overheads due to fall-through branches and conflict misses caused by the limited size of the tag memory. Large blocks cause large amounts of code to be loaded into the cache that is potentially evicted again without being executed. The average code block size should thus be chosen depending on the method cache size and its associativity in order to minimise both conflict misses and cache miss costs.

Since the method cache operates on variable sized code blocks, the cache hit rate is not suited to evaluate the performance of the cache. Large code blocks typically have a higher *hit rate* than smaller blocks as they have a higher chance of being used, but the *utilisation* of their code can be low, leading to a lower performance than when using smaller code blocks. Therefore, the ratio of bytes fetched

from the cache and bytes transferred from memory, the *transfer ratio*, is a better metric than the hit rate, but it does not account for performance advantages of large burst transfers over multiple smaller transfers. We thus use the total number of cycles (including stall cycles for memory transfers) as metric in our evaluation of various method cache and function splitter configurations.

To evaluate the impact of various region sizes, we compare the run time of the benchmarks when using large code blocks of up to 1 kB of size against using smaller code block sizes. The speedups in terms of cycles of smaller code blocks over large code blocks are presented as box plots, which each represent the results of the benchmarks for a particular function splitter configuration.

Figure 1 gives the results for a standard 4 kB FIFO method cache with a typical tag memory size of 32 entries. As seen in Table 9, the first setup (32/32) is equivalent to making almost all basic blocks a region. The advantages of loading nearly no unused code into the cache is thwarted by the tag memory size limitations and the overhead of additional branches though. The best performance can be achieved with code block sizes of around 128 to 256 bytes. In those cases the full size of the method cache is utilised to store code blocks (in contrast to smaller splittings), while minimising the overhead for jumps between code blocks. We also see that emitting large loops as single code blocks does not make any noticeable difference for our benchmarks. This is due to the fact that the benchmarks contain very few loops that meet all criteria of our heuristics.

When the size of the tag memory of the method cache is doubled without changing the size of the cache itself (Figure 2), smaller code blocks become more profitable, since more different code blocks can be stored in the cache. For function splitter setups emitting larger code blocks, the performance of the benchmarks is not limited by the tag memory size but the method cache size itself, though. Increasing the tag memory has thus no benefit.

To compare the performance of the method cache with a typical set-associative cache, we show the speedup of the method cache over a standard 4-way set-associative LRU instruction cache in Figure 3. For half of the benchmarks the method cache can achieve a performance that is at most 8% slower than with a much more expensive LRU instruction cache. In some cases the method cache can even outperform the LRU cache by taking advantage of fast large page-burst transfers.

5.3 WCET Evaluation

In order to evaluate the impact of the function splitter on the WCET performance, we used a novel *scope-based* cache analysis technique that has been developed within T-CREST and has been presented in [7]. In contrast to traditional data-flow based cache analyses which analyse the local cache state at each program point, the scope-based cache analysis looks at global properties of the cache state to detect cache conflicts. It is similar to persistence cache analyses, but generalises existing analysis techniques. It can thus be used with first-in-first-out (FIFO) caches and the method cache, in addition to least-recently-used (LRU) caches. The new analysis also detects more fine grained cache conflict scopes, improving the precision of the cache analysis.

Table 10 shows the precision of the analysis in comparison with a traditional state-of-the-art *persistence* analysis [8]. Since traditional analysis techniques only work well with LRU caches, we compare the analysis techniques using an LRU instruction cache. We used benchmarks from the

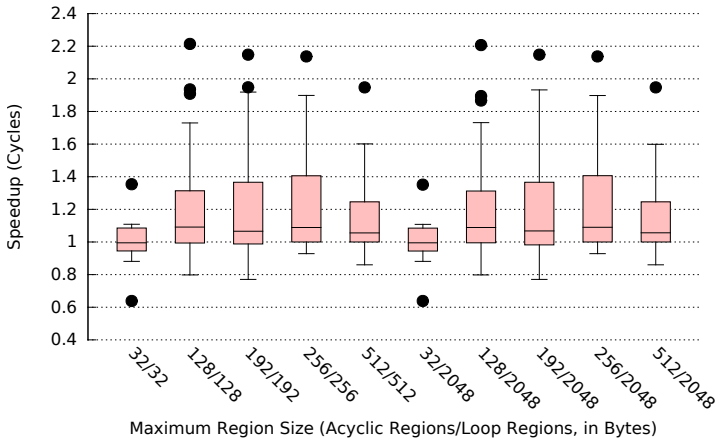

Figure 1: Speedup achived by splitting functions into smaller regions over a 1 kB region size setup on a 4 kB method cache and a tag memory of 32 blocks.

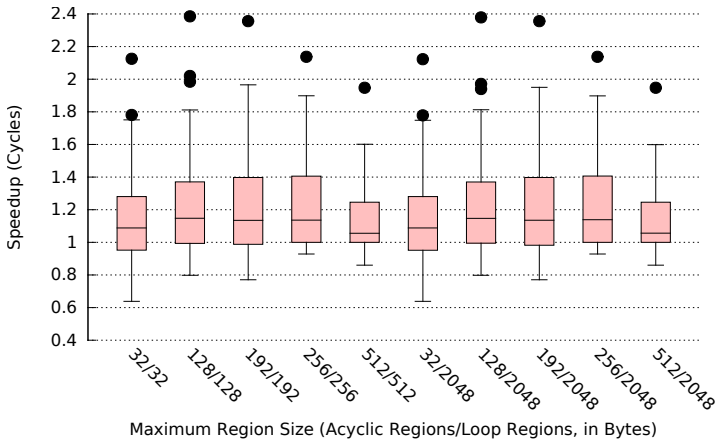

Figure 2: Speedup achived by splitting functions into smaller regions over a 1 kB region size setup on a 4 kB method cache and a tag memory of 64 blocks.

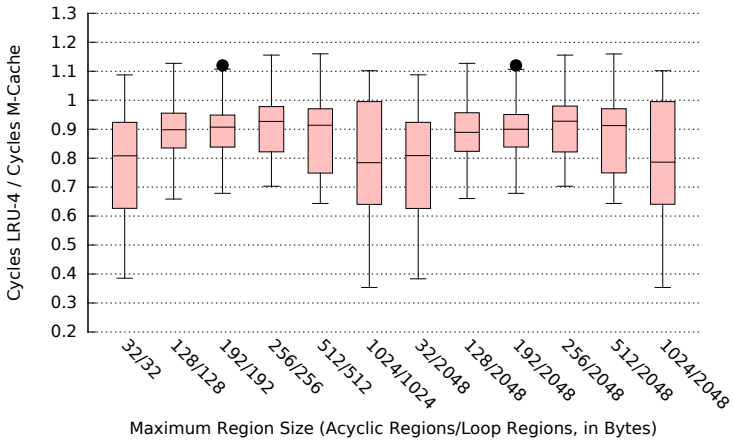

Figure 3: Speedup of a 4 kB 32 blocks method cache over a 4-way LRU instruction cache.

Mälardalen suite as well as from the PapaBench WCET benchmark[7] that execute at least 1 kB of code. Both the persistence analysis and the scope-based cache analysis have been implemented in the `platin` tool kit. Note that the composable nature of the Patmos memory hierarchy not only makes a global cache analysis possible in the first place, it also allows for a simple combination of analysis results from different analyses and tools.

The numbers in the columns of Table 10 give the number of WCET cycles relative to the persistence analysis (`pers`). The persistence analysis employs a data-flow analysis to detect cache conflicts. It is thus a path-sensitive analysis, making the analysis very precise. In contrast to our scope-based analysis, the persistence analysis is limited to LRU caches though. The column `simp` shows the performance of the scope-based cache analysis without the construction of the more precise conflict-free single-entry regions, it is thus limited to the same scopes as the persistence analysis, i.e., functions and loops. While the results of the scope-based region provide a higher estimate than `pers`, the results are sound for both a FIFO and an LRU cache though. The fourth column (`scope`) corresponds to the scope-based cache analysis with single-entry regions. It also applies to both FIFO and LRU caches. Therefore the results are also an upper bound for the loss of performance by using a FIFO cache instead of an LRU cache. Finally, the fifth column (`sim`) shows the measurement results for programs from the simulator.

As expected, the execution time of the simulation is equal or lower than the three static analysis results. This gives some confidence that the analyses are safe. We see that the simple scope-based analysis is equal to or gives more conservative results then the persistence analysis. However, the scope-based cache analysis with single-entry regions is equal to or outperforms the persistence analysis in all, except in three cases, even though it does not use a path-based conflict detection and is sound for both FIFO and LRU. For an LRU cache, the results of the scope-based analysis can be further improved by combining it with the more precise but also more computationally complex path-based conflict detection, thus outperforming the persistence analysis even more.

Finally, we can evaluate the WCET performance of the function splitter. We used the same setup as for the runtime evaluation (Section 5.2), using the Mälardalen WCET benchmarks that execute at least 4 kB of code. Figure 4 shows the WCET performance of a 4 kB variable-sided method cache and a tag memory of 32 entries relative to the WCET performance of a 4 kB 4-way set-associative cache depending on the function splitter configuration. The scope-based cache analysis was used to determine the WCET of the benchmarks in all setups. The results show a similar picture as in the runtime evaluation. Since no noticeable difference was found for a large region limit for loops, we only show the results using the same region grow threshold for both acyclic code and loops.

The best overall performance has been achieved with a region grow limit of 192 bytes. Only for a 256 byte grow limit some benchmarks can outperform the set-associative cache due to more efficient large page transfers, in the other cases the branch overhead is similar to or higher than the page-burst advantages. However, on average the method cache looses less than 10% to the set-associative cache in terms of WCET cycles, but requires only *one eighth* of the number of tags (assuming 16 byte cache lines). Since the scope-based cache analysis requires one analysis pass per cache set, the cache analysis of the method cache only requires *one pass* in comparison to 64 passes for the set-associative cache. It should be noted that since we used a non-path-sensitive scope-based cache analysis, those results are the same independent of the actual cache replacement policy used (either FIFO or LRU).

[7] `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`

	Benchmark	pers	simp	scope	sim
1	ndes	1.00	1.10	1.01	0.99
2	jfdctint	1.00	1.05	1.00	1.00
3	cnt	1.00	1.03	1.00	1.00
4	fdct	1.00	1.08	1.00	1.00
5	adpcm	1.00	1.00	1.00	1.00
6	edn	1.00	1.01	1.00	1.00
7	select	1.00	1.09	*0.63*	0.35
8	fbw/send_to_pilot	1.00	1.35	0.98	0.69
9	qsort	1.00	1.04	*0.83*	0.72
10	fbw/check_failsafe	1.00	1.22	*0.86*	0.82
11	fbw/check_values	1.00	1.22	*0.86*	0.80
12	ud	1.00	1.02	0.97	0.82
13	fbw/ppm_task	1.00	1.30	0.95	0.79
14	nsichneu	1.00	1.00	*0.45*	0.45
	1st Qu.	1.00	1.02	0.86	0.74
	Median	1.00	1.06	0.97	0.82
	3rd Qu.	1.00	1.19	1.00	1.00

Table 10: Analysis precision of scope-based cache analysis

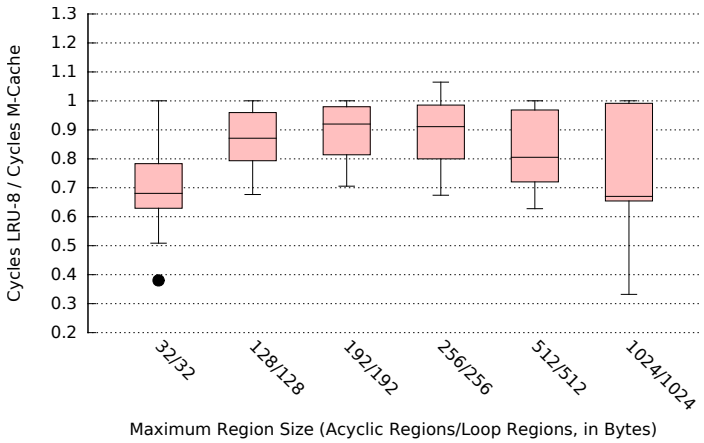

Figure 4: WCET speedup of a 4 kB 32 blocks method cache over a 4-way LRU instruction cache.

6 Full Compiler Tool Chain Evaluation

In this section we evaluate the impact of different compiler optimisation options on the generated code. For this evaluation, we again use the Mälardalen benchmark suite. We compare both run time measurements gained from the `pasim` simulator as well as WCET bounds calculated by AbsInt `aiT`.

Table 11 shows the impact of using the stack cache on the performance of the benchmarks. The table gives the measured runtime and analysed WCET bound of the individual benchmarks on a setup using a 2 kB stack cache and a 2 kB data cache relative to a setup using a single 4 kB data cache. In the setup using only a data cache, all writes to the stack have to go to the global memory. Using the stack cache eliminates the delays of those writes, but introduces additional delays if stack frames need to be written back to global memory or filled from global memory.

In about 40% of the benchmarks, the compiler is not able to put any data onto the stack cache, either because the benchmarks do not use stack allocated data or because the stack frames are dynamically allocated. No data cache accesses are eliminated in those cases. In almost all other cases, utilising the stack cache reduces the runtime and the WCET by several percent. For the `adpcm`, `expint` and `qurt` benchmarks the compiler is able to transfer nearly all data accesses to the stack cache, reducing the runtime by 8 to 10 percent. However, for the `fdct` benchmark we can observe an increase of the WCET by 17%. Enabling the use of the stack cache in this case has an adverse effect on other compiler optimisations, preventing the compiler from eliminating some memory accesses compared to the setup without a stack cache. The speedup gained by the use of the stack cache is eliminated by the costs of the additional data cache accesses to heap-allocated data.

Cache misses in the data cache are mostly compulsory misses and conflict misses in all the Mälardalen benchmarks, reducing the size of the data cache to only 2 kB in the setup without a stack cache has nearly no impact on the runtimes and WCET bounds of the benchmarks.

Patmos has been designed to include a dual-issue pipeline, i.e., it can execute up to two instructions in a single cycle. This requires the compiler to bundle instructions that should be executed simultaneously. In order to save hardware resources, the processor also has a single-issue configuration. The compiler supports both configurations by allowing the developer to select whether instructions should be scheduled for both pipelines or only for the first pipeline. The *Dual-Issue* column in Table 12 gives the simulated runtime and the WCET for the Mälardalen benchmarks for a dual-issue setup relative to a single-issue setup. Depending on the benchmarks, we observe speedups up to 26%, with an average speedup of 7%. The *fdct* benchmarks however shows a lower WCET performance for dual-issue than in the single-issue setup. This is due to the non-optimal nature of the bottom-up VLIW scheduler algorithm used for selecting the instruction bundles.

Patmos supports both delayed and non-delayed versions of all control flow (CFL) instructions. The delayed CFL instructions, introduced in the initial Patmos design, require the compiler to fill all delay slots with some instructions. If the compiler is not able to find such instructions, NOP instructions must be inserted, increasing the instruction cache costs. The newer non-delayed instructions do not have any delay slots but speculatively execute the instructions after the branch. If the branch is not taken, the branch has no penalties. If the branch is taken, the speculatively executed instructions are disposed, execution resumes at the branch target after the branch delay.

Benchmark	Sim	WCET	D$ Hits	D$ Misses	D$ Accesses
adpcm	0.93	*0.92*	0.02	0.97	0.02
bsort100	1.00	*1.00*	1.00	1.00	1.00
cnt	1.00	*1.00*	1.00	1.00	1.00
compress	0.93	*0.88*	0.63	0.66	0.64
cover	1.00	*1.00*	1.00	1.00	1.00
crc	1.00	*1.00*	0.97	0.99	0.98
edn	0.98	*0.99*	0.98	0.98	0.98
expint	0.93	*0.92*	0.00	0.00	0.00
fdct	0.97	*1.17*	0.94	1.84	1.23
fft1	0.86	*0.89*	0.00	0.00	0.00
fir	1.00	*1.00*	1.00	1.00	1.00
insertsort	1.00	*1.00*	1.00	1.00	1.00
janne_complex	1.00	*1.00*	1.00	1.00	1.00
jfdctint	0.98	*0.99*	0.93	0.98	0.96
lcdnum	1.00	*1.00*	1.00	1.00	1.00
lms	0.91	*0.93*	0.09	0.85	0.15
ludcmp	0.86	*0.85*	0.13	0.44	0.20
matmult	1.04	*1.00*	1.00	1.00	1.00
minmax	0.85	*0.90*	0.69	0.80	0.71
minver	0.85	*0.82*	0.21	0.44	0.28
ndes	1.06	*1.00*	1.00	1.00	1.00
ns	1.00	*1.00*	1.00	1.00	1.00
nsichneu	1.00	*1.00*	1.00	1.00	1.00
qsort-exam	0.99	*1.00*	0.83	0.93	0.87
qurt	0.87	*0.90*	0.02	0.11	0.02
select	0.98	*0.98*	0.74	0.79	0.76
statemate	0.78	*0.76*	0.83	0.67	0.80
ud	0.91	*0.91*	0.52	0.90	0.66

Table 11: Runtime, WCET and worst-case data cache accesses using a 2 kB stack cache and 2 kB data cache compared to a setup using a single 4 kB data cache.

	Dual-Issue		Non-Delayed CFL		Mixed CFL	
Benchmark	Sim	WCET	Sim	WCET	Sim	WCET
adpcm	0.77	*0.74*	1.01	*0.81*	0.99	*0.82*
bsort100	0.91	*0.93*	1.03	*0.89*	0.93	*0.87*
cnt	0.94	*0.95*	1.01	*1.00*	1.00	*1.00*
compress	0.95	*0.96*	0.98	*0.90*	0.95	*0.94*
cover	0.99	*0.99*	1.11	*0.60*	0.98	*0.72*
crc	0.94	*0.95*	1.02	*0.97*	1.00	*0.99*
edn	0.76	*0.92*	1.11	*1.01*	0.93	*1.00*
expint	0.78	*0.74*	1.01	*0.80*	0.99	*0.81*
fdct	0.91	*1.10*	1.02	*1.00*	1.00	*1.20*
fft1	0.91	*0.93*	0.95	*0.88*	0.95	*0.93*
fir	0.75	*0.91*	1.20	*1.00*	0.99	*0.99*
insertsort	1.00	*1.00*	0.99	*0.93*	0.99	*0.93*
janne_complex	0.95	*0.95*	0.94	*0.76*	0.92	*0.76*
jfdctint	0.89	*0.93*	1.01	*0.98*	1.00	*1.00*
lcdnum	0.93	*0.91*	0.94	*0.79*	0.91	*0.82*
lms	0.94	*0.95*	0.95	*0.85*	0.94	*0.91*
ludcmp	0.93	*0.94*	0.94	*0.87*	0.95	*0.93*
matmult	0.77	*0.90*	1.24	*1.02*	1.00	*1.00*
minmax	0.97	*0.97*	0.97	*0.88*	0.94	*0.92*
minver	0.91	*0.92*	0.93	*0.87*	0.93	*0.92*
ndes	0.97	*0.97*	1.02	*0.97*	1.06	*0.99*
ns	0.98	*0.99*	1.15	*0.89*	0.86	*0.92*
nsichneu	1.00	*1.00*	0.68	*0.76*	0.68	*0.76*
qsort-exam	0.92	*0.94*	0.86	*0.74*	0.87	*0.81*
qurt	0.92	*0.94*	0.95	*0.88*	0.95	*0.93*
select	0.92	*0.96*	0.93	*0.79*	0.90	*0.85*
statemate	0.88	*0.89*	0.94	*0.92*	0.94	*0.94*
ud	0.84	*0.86*	0.99	*0.86*	0.97	*0.89*

Table 12: Runtime and WCET by scheduling for dual-issue compared to single-issue, and by using non-delayed branches compared to using delayed branches only.

By default, the `patmos-clang` compiler emits non-delayed branches if it cannot fill a single delay slot, otherwise a delayed CFL instruction is used. The compiler can also be set up to emit only delayed or only non-delayed instructions. The runtime and WCET bounds achived by using either only non-delayed instructions or a mixture of delayed and non-delayed instructions are shown in Table 12, relative to using only delayed instructions. Using only non-delayed branches has a negative impact on the measured runtime of some of the benchmarks. The reduced code size however has a large impact on the worst-case instruction cache costs, reducing the WCET of the benchmarks by 12% on average. Using both delayed an non-delayed instructions (the *mixed CFL* setting) leads to a better average case performance for our benchmarks, but reduces the WCET only by 9% on average compared to using only delayed instructions.

Table 13 finally shows the runtime and WCET for the Mälardalen benchmarks for the predefined optimisation levels `O1` and `O2` compared to compiling at level `O0`. At `O0`, the compiler stores all variables and results in global memory in order to enable debugging. At `O1`, most optimisations related to memory accesses, register allocation and scheduling are enabled. In particular, most variables are moved to registers, leading to huge performance gains. Level `O2` enables some more complex loop transformations and inlining. However, these optimisations only affect the performance by a few percent, and can have adverse effects on some benchmarks.

The Patmos compiler tool chain links applications at bitcode level by default. This enables the compiler to perform additional whole-program optimisations after linking. The last column in Table 13 shows the performance of the benchmarks when link-time optimisations are disabled (`-fpatmos-skip-opt`) and modules are optimised separately at level `O2`. Compared with the results at `O2`, the performance of some benchmarks is actually decreased by the link-time optimisations. This is due to the function inliner, which has not been tuned to the properties of the method cache. By duplicating code, the function inliner increases the code size, leading to more cache conflicts. The advantages of inlining and specialising code at call sites is not sufficient to counter the higher cache miss costs caused by the inliner.

While LLVM provides a `O3` optimisation level, the results for this level have been omitted since they are identical to the results at `O3`. Only an argument promotion optimisation is executed at `O3` in addition to the optimisations performed at `O2`, which has no effect on our benchmarks.

	-O1		-O2		-fpatmos-skip-opt	
Benchmark	Sim	WCET	Sim	WCET	Sim	WCET
adpcm	0.20	*0.13*	0.20	*0.13*	0.09	*0.06*
bsort100	0.29	*0.24*	0.29	*0.24*	0.27	*0.23*
cnt	0.19	*0.20*	0.17	*0.20*	0.21	*0.22*
compress	0.29	*0.25*	0.30	*0.26*	0.44	*0.46*
cover	0.11	*0.05*	0.11	*0.05*	0.09	*0.04*
crc	0.18	*0.17*	0.10	*0.12*	0.12	*0.16*
edn	0.17	*0.25*	0.16	*0.24*	0.17	*0.25*
expint	0.21	*0.14*	0.21	*0.14*	0.09	*0.07*
fdct	0.18	*0.19*	0.18	*0.19*	0.21	*0.21*
fft1	0.29	*0.29*	0.32	*0.30*	0.26	*0.25*
fir	0.06	*0.14*	0.06	*0.14*	0.04	*0.09*
insertsort	0.30	*0.26*	0.30	*0.26*	0.32	*0.26*
janne_complex	0.16	*0.15*	0.16	*0.15*	0.15	*0.12*
jfdctint	0.28	*0.28*	0.28	*0.28*	0.29	*0.27*
lcdnum	0.37	*0.36*	0.37	*0.36*	0.37	*0.30*
lms	0.36	*0.36*	0.36	*0.35*	0.23	*0.23*
ludcmp	0.51	*0.50*	0.51	*0.49*	0.33	*0.31*
matmult	0.18	*0.24*	0.18	*0.24*	0.17	*0.22*
minmax	0.40	*0.46*	0.40	*0.46*	0.42	*0.38*
minver	0.29	*0.24*	0.29	*0.23*	0.26	*0.30*
ndes	0.17	*0.25*	0.18	*0.25*	0.24	*0.28*
ns	0.18	*0.24*	0.18	*0.24*	0.18	*0.22*
nsichneu	0.39	*0.51*	0.39	*0.51*	0.31	*0.40*
qsort-exam	0.33	*0.34*	0.33	*0.34*	0.22	*0.20*
qurt	0.49	*0.49*	0.49	*0.49*	0.31	*0.32*
select	0.34	*0.37*	0.34	*0.37*	0.23	*0.25*
statemate	0.06	*0.06*	0.05	*0.05*	0.51	*0.57*
ud	0.27	*0.24*	0.28	*0.23*	0.15	*0.13*

Table 13: Runtime and WCET by `patmos-clang` optimisation levels relative to `-O0`.

7 Requirements

For the sake of completeness, we 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. NON-CORE and FAR requirements are not listed here.

7.1 Industrial Requirements

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

The compiler supports inline assembly, which can be used to implement storing and restoring threads.

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.

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 utilise 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.

The documentation of the tool chain can be found in the Patmos handbook in the patmos repository. Additionally, all tool chain source repositories contain a `README.patmos` ***file, which explains how to build and use the tools provided by the repository.*** *Further information about the LLVM compiler can be found in the LLVM user guide.*[8]

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

7.2 Technology Requirements

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

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

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. ***Additionally, loop bounds and other flow constraints can be annotated by special `#pragmas`, and passed down the compiler tool chain and exported to the WCET analysis tool.***

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. ***Additionally, loop bounds and other flow constraints can be annotated by special `#pragma`s, and passed down the compiler tool chain and exported to the WCET analysis tool.***

8 Conclusion

This report provided an evaluation of the full compiler, focussing on aspects that have not been evaluated in previous reports. First, we presented a summary of all the features in the compiler that required special attention and support.

The single-path code generator has been evaluated focussing on qualitative aspects of the transformation. Automated generation of single-path code can be successfully performed. The quality and performance of the resulting code is highly dependent on the semantic properties and coding style used. In conjunction with the composable Pamtos architecture, worst-case bounds can be easily determined, and stable execution times can be obtained. We have observed that there is room for improvement w.r.t. execution costs that could be addressed by specific tailored compiler code transformations, which we will address in future research efforts.

The function splitter has been evaluated using the Patmos simulator and the scope-based cache analysis for deriving WCET bounds. We identified a function splitter configuration producing code blocks of up to 192 bytes as the optimal setup for a typical method cache configuration.

The full compiler evaluation has shown that the architectural features that reduce resource sharing and the number of memory accesses have the most benecifial impact on the (worst-case) performance.

References

[1] 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.

[2] Florian Brandner, Stefan Hepp, and Alexander Jordan. Criticality: static profiling for real-time programs. *Real-Time Systems*, 50(3):377–410, 2014.

[3] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *Proc. of the Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, 2014.

[4] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[5] 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 *Proc. of the Int. Workshop on Workload Characterization*, WWC '01, pages 3–14. IEEE, 2001.

[6] Stefan Hepp and Florian Brandner. Splitting functions into single-entry regions. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2014). to appear*, 2014.

[7] Benedikt Huber, Stefan Hepp, and Martin Schoeberl. Scope-based method cache analysis. In *Int. Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASIcs*, pages 73–82. Schloss Dagstuhl, 2014.

[8] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.

[9] Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 55–64, New York, NY, USA, 2013. ACM.

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

[11] Daniel Prokesch, Benedikt Huber, and Peter Puschner. Towards Automated Generation of Time-Predictable Code. In *Int. Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASIcs*, pages 103–112. Schloss Dagstuhl, 2014.

[12] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 2013.