The platin Tool Kit - The T-CREST Approach for Compiler and WCET Integration

Stefan Hepp[1], Benedikt Huber[2], Jens Knoop[1], Daniel Prokesch[2], and Peter Puschner[2]

[1] Institute of Computer Languages
Vienna University of Technology
Vienna, Austria
{hepp,knoop}@complang.tuwien.ac.at
[2] Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
{benedikt,daniel,peter}@vmars.tuwien.ac.at

Abstract. The construction of safety-critical real-time applications requires predictable computer platforms that enable a safe and tight static analysis of those systems. The worst-case performance and the availability of tight bounds on the worst-case execution time (WCET) of the tasks of the applications are of central importance in such systems.
The compiler tool-chain plays an integral part in a real-time platform infrastructure. Not only is the compiler responsible for relating source code level annotations such as flow facts to the generated machine code – a task necessary to achieve high-quality bounds on optimised code. Also, the analysis tools profit from program information that is readily available in the compiler but difficult to retrieve from the generated binary alone. Therefore, the compiler should export internal knowledge about the program to the analysis. Furthermore, compilation for hard real-time systems requires optimisations that specifically aim at reducing the worst-case execution path instead of reducing the average case performance. This requires feedback from the worst-case analysis back to the compiler.
In this paper we describe the our approach to the compiler tool integration that has been realised in the platin tool kit, developed in the EU FP7 T-CREST project. The platin tool kit is a portable glue tool that interfaces our LLVM-based T-CREST compiler with several research and industrial strength analysis tools. Our approach is transferable to other compiler tool-chains and minimises the effort for adapting them for the requirements of real-time platforms.

1 Introduction

Embedded computer systems are playing an increasingly important role in applications that are time-critical, e.g., in fly-by-wire applications, in medical equipment, and in control systems of nuclear power plants. To ensure safety, the computer systems controlling the actuators in these applications have to respond

to changes in the environment within strict time bounds. It is thus important to design and implement these systems to meet their timing constraints and to show that the implementation indeed fulfils all timing requirements. Despite the stringent timing requirements of these time-critical applications, the importance of *time as a first-order property of embedded systems behaviour* is not adequately reflected by the platforms and methods/tools widely used for the construction of the embedded computer systems for these applications.

Within the T-CREST project a new embedded multi-core platform was developed [14], which emphasised time-predictability in all design decisions. The T-CREST platform consists of a novel processor core called Patmos [13], a time-predictable network-on-chip (NoC) that connects the cores to each other and to a predictable memory controller, and a tool chain centred around LLVM [9] for compiling and analysing applications written in C code [11].

The primary task of the compiler tool chain in such a platform is to generate machine code for the target architecture. However, the compiler should also to try to minimise the worst-case execution time (WCET) of the application tasks, and support the worst-case analysis tools in finding tight and safe WCET bounds. This requires interaction of the compiler with the WCET analysis tools. In the T-CREST project we implemented common routines for tool integration and analysis in a separate tool kit called *platin* that requires only small adoptions of the compiler and can be reused for other target architectures as well. The platin tool kit is centred around its native *PML* file format that stores information about the program structure and meta-information such as flow facts and analysis results in a target-machine agnostic form. The tool kit not only contains tools to interface with external analysis tools such as the industry-standard AbsInt aiT WCET analyser, it also provides tools for tasks such as flow fact transformation, WCET analysis, graph visualisation and tool configuration.

In this paper we give an overview of the T-CREST platform, its compiler tool chain and the platin tool kit. We present how the platin tool kit binds the compiler and the WCET analysis tools together, show how to use use the platin tool kit, and briefly discuss the steps required to adapt platin for a new target architecture. The rest of the paper is structured as follows. Section 2 introduces the T-CREST platform, while Section 3 overviews the Patmos compiler tool chain. Section 4 presents the platin tool kit its interaction with the compiler and analysis tools, and overviews the tools provided by platin. Section 5 demonstrates the use of platin by means of an example. We discuss related work in Section 6 and conclude this paper with Section 7.

2 The T-CREST Platform

The goal of the T-CREST project[3] was to develop a fully time-predictable multi-core platform. The T-CREST platform consists not only of a novel processor

[3] Results and publications of the T-CREST project are available from the project website `http://www.t-crest.org/`

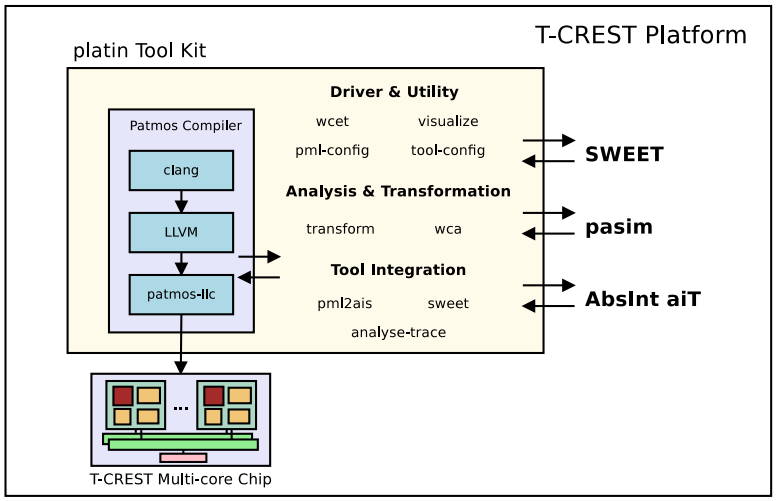


Fig. 1: The T-CREST platform and its compiler and analysis software ecosystem.

core, but also includes a time-predictable memory system, a compiler, analysis tools and runtime libraries. All of them are designed to play together to achieve a highly predictable platform for embedded systems.

The hardware side of the T-Crest platform consists of the T-Crest multi-core *chip*. It contains a configurable number of *processor cores*. Each core contains a Patmos *processor* and several local memories. The Patmos processor [13] uses a fully-predicated 32-bit RISC-style instruction-set architecture (ISA) and a five-stage in-order pipeline. Each core features a data cache, a stack cache [1], a method cache [2] and a local scratchpad. The cores are connected to each other by a time-predictable network-on-chip [7] called Argo, which can be used for message passing. A separate time-predictable memory interconnect [4] called Bluetree connects the cores to the memory controller [5] for the shared RAM.

Figure 1 gives a software-centric overview of the T-CREST platform. The LLVM-based Patmos *compiler tool chain* uses the clang C frontend to parse C code into LLVM bitcode, which is then optimised by LLVM bitcode passes. The LLVM backend for Patmos `patmos-llc` generates machine code for the Patmos processor. The platin tool kit [11] is a key component in the T-Crest platform for tool integration and WCET analysis. It is tightly coupled with the compiler tool chain and serves three main tasks. First, it provides tools for flow fact transformation and for program analysis. Second, it interfaces with existing analysis tools in order to bring their analysis functionality to the T-Crest platform. Among the supported tools are the SWEET flow analyser and the industry-standard AbsInt aiT WCET analyser. Platin can also be used to analyse execution traces generated by the Patmos simulator `pasim`. Third, platin provides utility tools for result visualisation and tool configuration, as well as

driver tools that chain multiple analysis and transformation steps into single, easy to use commands.

We will overview the Patmos compiler tool chain in the next section, while the rest of this paper presents the platin tool kit in more detail.

3 The Patmos Compiler

The Patmos processor developed within T-CREST is designed for high time predictability [13]. The architectural features of this processor are designed to improve performance yet remain inherently timing analysable. This is achieved by using static (compile-time) alternatives for commonly used performance-enhancing features at runtime in order to reduce hard-to-analyse dynamic behaviour. A worst-case timing analysis tool can then be used to derive tight WCET bounds for the real-time tasks of the embedded application.

The task of the Patmos compiler is thus twofold. First, the compiler must generate code that targets the Patmos ISA and exerts control over the components of the processor core so that the generated program exhibits a low WCET [2,1]. Second, the compiler must support the WCET analysis by providing information available in the compiler that usually is discarded but is valuable for automated and precise timing analysis. This includes preserving information about the control-flow structure, but also flow annotations provided by the user that constrain the possible flow of control, e.g., bounds on the maximum number of loop iterations (*loop bounds*). The compiler in turn can profit from static analysis results from the timing analysis to guide optimisations towards a good worst-case performance. This requires an integration of the compiler and the WCET analysis tools.

Figure 2 gives an overview of the compiler tool chain. The compiler is based on the LLVM compiler framework [9]. At the beginning of the compilation process, each C source code file is translated to LLVM intermediate representation (*bitcode*) by the C frontend `clang`. The user application code as well as standard C libraries and runtime support libraries are linked on this intermediate level by the `llvm-link` tool, presenting subsequent analysis and optimisation passes as well as the code generation backend a complete view of the whole program. This control-flow graph (CFG) oriented intermediate representation is particularly suitable for generic target independent optimisations, such as common subexpression elimination, which are readily available through the LLVM `opt` tool. The `llc` tool constitutes the compiler backend. It translates LLVM bitcode into machine code for the Patmos ISA, addressing the target-specific features for time predictability. The backend produces a relocatable ELF binary containing symbolic address information, which is processed by `gold`,[4] defining the final data and memory layout, and resolving symbol relocations. An important property of this compilation flow stems from the fact that the application is already linked at intermediate level: Optimisations and the code generator have a complete

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

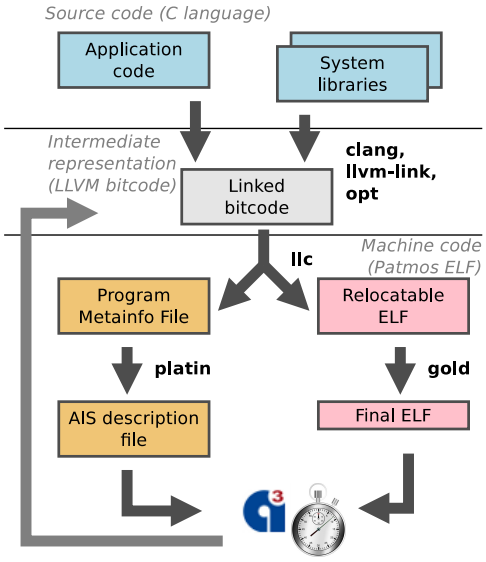


Fig. 2: Overview of the Patmos compiler tool chain

view of the program, which is necessary for optimisations that need to balance the use of a shared resource across the whole program execution. For example, Patmos' specialised software-controlled caches require the compiler to be aware of all cache accesses along the worst-case path for it to be able to generate code that exhibits lowest possible WCET.

In addition to the machine code, the backend exports complete information about the control-flow structure of both bitcode and machine code as well as information about the program obtained by the compiler in the Program Metainfo Language (PML) format, as detailed in the following section. The `platin` tool kit uses these PML files to perform analysis tasks and to transform flow facts. It is also able to export program information to analysis tools such as the AbsInt WCET analysis tool `aiT`. Analysis results are imported back into the PML file, which can in turn be passed back to the compiler for iterative WCET driven optimisation.

The platform, including the processor, a simulator, the compiler tool chain including the platin tool kit as well as a set of benchmarks is available as open-source from the T-Crest organisation at github.[5] The Patmos handbook [12] provides detailed information about the installation, a description of the processor core and its instruction set architecture (ISA) and documents the use of the compiler tool chain.

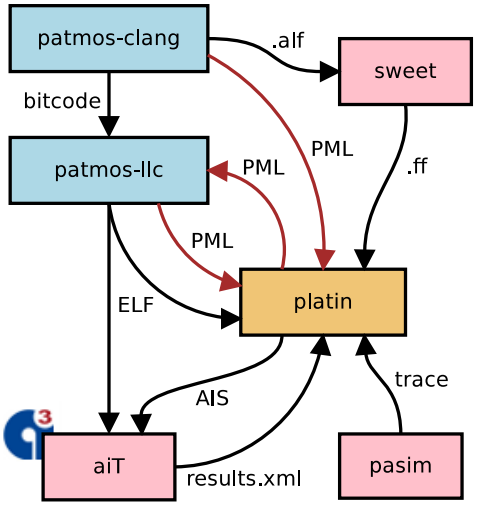


Fig. 3: The platin tool interacts with the compiler with its native PML file, while it communicates with other tools using exporters and importers for their file formats.

4 The Platin Tool Kit

The main task of the platin tool kit is the tool integration in the T-Crest platform. It uses a YAML[6] based file format called PML as its native file format, which stores control flow information, flow facts and value facts, analysis results, information to relate different code representations and a hardware description. Apart from performing tool integration tasks, the platin tool kit has been extended to provide tools not only for visualisation and inspection of information stored in PML files but also to include its own set of cache and path analyses to perform a WCET analysis. Section 4.2 overviews platin's main tools.

Platin's interaction with other tools is shown in Figure 3. It gathers information from a number of sources. The LLVM backend provides the control-flow and call targets on both bitcode and machine-code level. The LLVM PML exporter also retrieves value facts about data pointers as well as flow facts from LLVM-internal analysis passes and adds them to the generated PML file. The platin tool kit also contains several analysis drivers for external analysis tools. The trace analysis tool derives execution timings and flow facts from a simulator's execution trace (`pasim` for Patmos). Platin can also use the abstract interpretation based analyser SWEET [10] to find additional flow facts. An LLVM plugin exports the LLVM bitcode representation of the program to the Artist

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

[6] A "human friendly data serialization standard for all programming languages", see `http://yaml.org/`

Flow Analysis Language (ALF) that SWEET uses as input language. Flow facts found by SWEET are then imported by platin and mapped back to bitcode.

Using the information provided by the compiler, platin is able to transform the gathered information from bitcode level to the machine code level. The combined set of flow facts and value facts is then passed on the AbsInt WCET analyser aiT in aiT's native AIS format, or used in the internal WCET analysis tool called WCA. A flow fact simplification step ensures that the flow facts are expressed in a form that is understood by the used WCET analysis tool.

4.1 Flow Fact Transformation

LLVM splits the task of compiling source code to machine code into two major steps: compiling to and optimising on a machine-independent intermediate representation called bitcode (`clang`) and lowering bitcode down to machine code using machine specific optimisations in the backend (`llc`). The LLVM backend does not change the control flow in a major way since optimisations such as loop transformations and inlining are performed at bitcode level. This enables platin to map bitcode and machine code across the backend automatically using relation graphs [6] with almost no adaption of the backend. Using these relation graphs, platin is able to transform linear flow facts from bitcode to machine code without further user assistance.

Maintaining flow facts across high-level optimisations is inherently more difficult and requires at least some compiler support. There are various approaches to that problem. Transforming the flow facts along with the optimisation transformations can be done either by the compiler itself as implemented in the WCC compiler [3], or by an external tool that requires a log of all compiler transformations as proposed by Kirner et al˙ [8]. Platin leaves the task of high-level flow fact transformation to the compiler. The Patmos compiler must therefore ensure that flow facts that are exported to PML match the exported optimised bitcode. Flow facts that are derived directly from LLVM analyses do not need to be co-transformed since the LLVM framework itself either updates or reruns analyses after optimisation passes as required. For manual source code annotation, the Patmos compiler currently supports constant loop bound flow facts as source code pragmas. The compiler disables transformations that might invalidate these loop bounds for functions containing such source annotations. The preserved source code pragmas can thus be directly exported to PML.

For the future we plan to support arbitrary linear flow facts in the Patmos compiler by using source code markers. In contrast to other techniques, using flow markers requires only minimal changes to the compiler. In particular, optimisations do not need to update flow facts as long as the code transformations preserve the sequence of markers on any program path. This makes integrating and maintaining flow fact support in a large existing and constantly evolving compiler such as LLVM much more feasible. Support for source code markers is still under development in the Patmos compiler though.

4.2 Platin Tools

The core of platin is a Ruby framework for working with PML data. It provides common functionality such as reading and writing PML files, accessing and traversing PML data structures, merging and modifying PML data, constructing various graph representations and working with context-sensitive information. The tools and analyses in platin are built on top of that framework. The tools typically accept one or more PML input files and a number of options, and will generate a new PML output file. The tools can be chained together by passing the output PML file of any tool as input of another platin tool. Platin tools can also invoke other platin tools internally in order to implement complex functionality. In this case, PML data is passed between the tools in-memory. The Ruby scripting language enables rapid development of tools and analyses and allows the developer to focus on the task at hand, which is especially essential in a research environment. While a Ruby implementation implies some performance drawbacks compared to other languages, we did not find the performance of the platin tools to be an issue in our experiments.

Platin provides several tools to work with its native PML file format. The platin `pml` tool can merge and validate PML files or print out flow facts, value facts and timing analysis results in a condensed form. The `visualize` tool can be used to visualise control flow graphs and relation graphs.

Platin can also be used for for tool configuration. It uses PML files to configure parameters of the hardware model, such as cache parameters and memory latencies. The platin `pml-config` tool can be used to generate or modify such a PML hardware model, while the `tool-config` tool generates command line options for tools like the compiler and the simulator to configure them consistent with the hardware model. Other tools like the WCET analysis tool and the aiT export also use the PML hardware model configuration to setup the timing parameters.

For tool integration, platin provides tools such as `sweet`, `analyze-trace` and `pml2ais`. The `sweet` tool invokes SWEET to find flow facts. The results are parsed and added to the PML file. The `analyze-trace` tool generates flow facts from simulation runs, which are only valid for the inputs used in the simulation but are useful for testing the correctness and precision of WCET analyses. Flow facts are attached either at bitcode or at machine code level, depending on their source. The `transform` tool converts flow facts between different levels. The `pml2ais` tool in turn exports flow facts to the AbsInt aiT AIS file format and generates an analysis project file for aiT based on the platin configuration.

The platin `wcet` tool is a driver tool for the WCET analysis tools. It invokes either AbsInt aiT using the `pml2ais` exporter or platin's internal WCET analysis `wca`. The `wcet` tool can optionally use many of the above tools to find and transform additional flow facts. It also sets up the analysis tools according to the PML hardware model and provides options to configure specific analysis modes such as always-hit or always-miss cache analyses.

4.3 Integrating platin Into Other Compiler Tool Chains

The platin tool kit has been designed to support multiple architectures with a minimal effort for adapting platin and the compiler tool chain. The PML file format and most of the functionality of platin is architecture independent. Memory latencies and caches are configured in a generic hardware model. Support for analysis tools like aiT and SWEET that support multiple target platforms is implemented in generic platin tools. Architecture dependent analysis and tool integration code is encapsulated in architecture modules in platin. Adapting platin to a new architecture thus only requires the implementation of a new architecture module for that platform, which invokes platform-specific tools such as a simulator and performs basic analysis tasks such as deriving the WCET of a basic block.

Platin requires a compiler backend that generates PML files. For LLVM backends, the PML export machine-function pass can be reused, as it also has been implemented in a generic way. This is possible due to the generic representation of machine code in LLVM backends. The PML export pass creates PML files, exports the structure of machine code, bitcode and relation graphs. Only the classes that retrieve target-specific information such as call or jump targets and interface with backend analysis passes need to be specialised. Work on high-level support for flow-fact transformation in the `clang` frontend and on bitcode level can be reused directly from the Patmos compiler, since the frontend and middle-end is platform independent.

Platin fully supports the Patmos platform and has some initial support for an ARM tool chain support. We do believe that basic support for other LLVM based compiler tool chains can be achieved comparatively quickly, as only a few key components in the LLVM backend and in platin need to be implemented or adapted. As a result and due to platin's open source nature, the platin tool kit can be useful for other projects in the domain of embedded real-time systems as well.

5 Example

In this section we demonstrate some of the tools of `platin`. We show a typical workflow by compiling and analysing a small demo application on Patmos. A quick start guide for installing the Patmos tool chain can be found in the Readme file of the Patmos repository[7] or in the Patmos handbook [12].

Listing 1 shows the content of `sort.c`. It contains a simple insertion sort implementation in function `sort`. Our target function for analysis is `gen_sort`, which fills an array with N pseudo-random numbers and then sorts the array. In order to prevent the compiler from inlining and removing our analysis target function, we mark the function as `noinline`. The code contains loop bound annotations for the WCET analysis in the form of pragmas.

[7] `https://github.com/t-crest/patmos`

Listing 1: Demo application that initialises and sorts an array.

```
#include <stdlib.h>

#define MAX_SIZE 100

void sort(int *arr, size_t N) {
  #pragma loopbound min 0 max 99
  for (int j = 1; j < N; j++) {
    int i = j - 1;
    int v = arr[j];
    #pragma loopbound min 0 max 99
    while (i >= 0 && arr[i] >= v) {
      arr[i+1] = arr[i];
      i = i - 1;
    }
    arr[i+1] = v;
  }
}
void gen_sort(int *arr, size_t N) __attribute__((noinline));
void gen_sort(int *arr, size_t N) {
  #pragma loopbound min 1 max MAX_SIZE
  for (size_t i = 0; i < N; i++) {
    arr[i] = rand() % N;
  }
  sort(arr, N);
}
int main(int argc, char** argv) {
  srand(0);
  int arr[MAX_SIZE];
  size_t N = rand() % (MAX_SIZE / 2) + (MAX_SIZE / 2);

  gen_sort(arr, N);

  return 0;
}
```

All tools in the Patmos tool chain are configured to use the default Patmos hardware configuration if no further options are given. In this example we show how to use `platin` to configure a different hardware setup. For this, we use `pml-config` to generate a modified hardware model:

```
platin pml-config --target patmos-unknown-unknown-elf \
                  -o config.pml -m 2k -M fifo8
```

This command generates a new `config.pml` file containing a description of the default hardware model, except that we use a method cache of only half the size (2 KB size with a tag memory of 8 entries).

In the next step, we compile our program using the `patmos-clang` compiler driver. We also use the `platin tool-config` tool to setup the compiler according to our modified hardware model. `tool-config` can be used in a similar manner to setup `pasim`, the Patmos simulator. We need to explicitly enable optimisations with `-O2`, as the default optimisation level is `-O0`.

```
patmos-clang `platin tool-config -i config.pml -t clang` \
             -O2 -o sort -mserialize=sort.pml sort.c
```

Listing 2: Analysis report for the `sort` application

```
---
- analysis-entry: gen_sort
  source: trace
  cycles: 49089
- analysis-entry: gen_sort
  source: platin
  cycles: 644867
  cache-max-cycles-instr: 651
  cache-min-hits-instr: 398
  cache-max-misses-instr: 3
  cache-max-cycles-stack: 0
  cache-max-misses-stack: 0
  cache-max-cycles-data: 436779
  cache-min-hits-data: 0
  cache-max-misses-data: 10599
  cache-max-stores-data: 10200
  cache-unknown-address-data: 20799
  cache-max-cycles: 437430
```

The driver calls all commands necessary to compile the source code, link and optimise the bitcode and generate and link the final binary `sort`. The option `-mserialize` causes the compiler to generate the PML file `sort.pml`. It contains a description of the application control flow at bitcode level (after the bitcode optimisations) and of the final machine code. It also contains value facts and flow facts such as loop bounds as found by the compiler as well as our source-code loop annotations, and relation graphs relating the bitcode and machine code control flow graphs.

Now we are ready to analyse our target function. We use the `platin wcet` driver tool to run all necessary commands, including the trace analysis and the platin WCET analysis tool WCA. The driver tool will automatically try to run the AbsInt aiT analysis tool if it is installed.

```
platin wcet -i config.pml --enable-trace-analysis --enable-wca \
            -b sort -e gen_sort -i sort.pml --outdir tmp \
            -o wcet.pml --report report.txt
```

We need to pass the name of the binary file (`-b`) and both the compiler generated PML file and the hardware model PML file (`-i`) to platin. The `-e` option tells platin the name of the analysis target function. The optional `--outdir` option causes platin to keep temporary files and store them in the given directory, mainly the generated project files for the AbsInt analyser tool `a3patmos`. The optional `-o` option stores detailed analysis results such as the found WCET bounds for the target function, execution timings of basic blocks and execution frequencies of blocks on the worst-case path along with the program information from the input files in a PML file for further analysis or for WCET-driven optimisations.

The `--report` option causes platin to store the result summaries of the analyses in `report.txt`. Listing 2 shows the content of that file. In this example the

Listing 3: Flow facts from LLVM and user annotations as reported by `platin`

```
=== flowfacts generated by llvm.bc ===
--- loop-bound ---
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: gen_sort/for.cond>:
  ↪ [1 gen_sort/for.cond] less-equal (1 + %N)>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: gen_sort/for.cond.i>:
  ↪ [1 gen_sort/for.cond.i] less-equal (1 umax %N)>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
=== flowfacts generated by user.bc ===
--- loop-bound ---
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/for.cond>:
  ↪ [1 gen_sort/for.cond] less-equal 101>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/for.cond.i>:
  ↪ [1 gen_sort/for.cond.i] less-equal 100>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/while.cond.i>:
  ↪ [1 gen_sort/while.cond.i] less-equal 100>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
```

platin WCET analysis derives a lower WCET bound than aiT. aiT is able to find better loop bounds and thus finds fewer data cache misses for the sort loop, but it assumes higher costs for instruction cache misses than platin.

Both analyses seem to highly over-approximate the actual WCET when compared to the trace results of the execution. However, while we assume that in the worst case the whole array is used, the actual execution only fills and sorts a fraction of the array. Hence the measured execution time is not a good indicator for the worst-case performance.

The inner loop of the sort function is a triangle loop. Our annotated global loop bound of $(N-1)^2$ is thus about a factor of two too large. For loops with constant bounds, LLVM is capable of detecting such triangle loops and deriving the correct bounds automatically. Our PML export uses the LLVM analysis results to generate additional flow facts. platin provides a tool to print all flow facts in a PML file in a compact form.

```
platin pml -i sort.pml --print-flowfacts
```

Listing 3 shows the output of that command. We find our manual loop annotations in the `user.bc` origin section. Note that LLVM inlined the `sort()` function, therefore our loops are now in function `gen_sort`.[8] The loop bounds are expressed as flow constraints on the loop header blocks.[9] We can also see that LLVM managed to find parametric loop bounds for two loops, but failed to find a loop bound for the inner triangle loop since in our case the size of the

[8] Function `__umodsi3` implements the modulo operator, as Patmos does not provide a modulo instruction in hardware.

[9] The right-hand side of the constraint is larger than our loop bound by one because the loop header is executed one additional time more than the loop body to jump out of the loop when the loop condition becomes false.

array to sort is not fixed but parametric. It is thus necessary to annotate the inner loop manually. Platin supports arbitrary linear flow constraints in PML. It is possible to manually supply additional flow constraints in PML format. Support for source code flow annotations beyond local loop bounds in the Patmos compiler is planned for future development.

We can also use platin to visualise control-flow graphs, call-graphs and relation graphs:

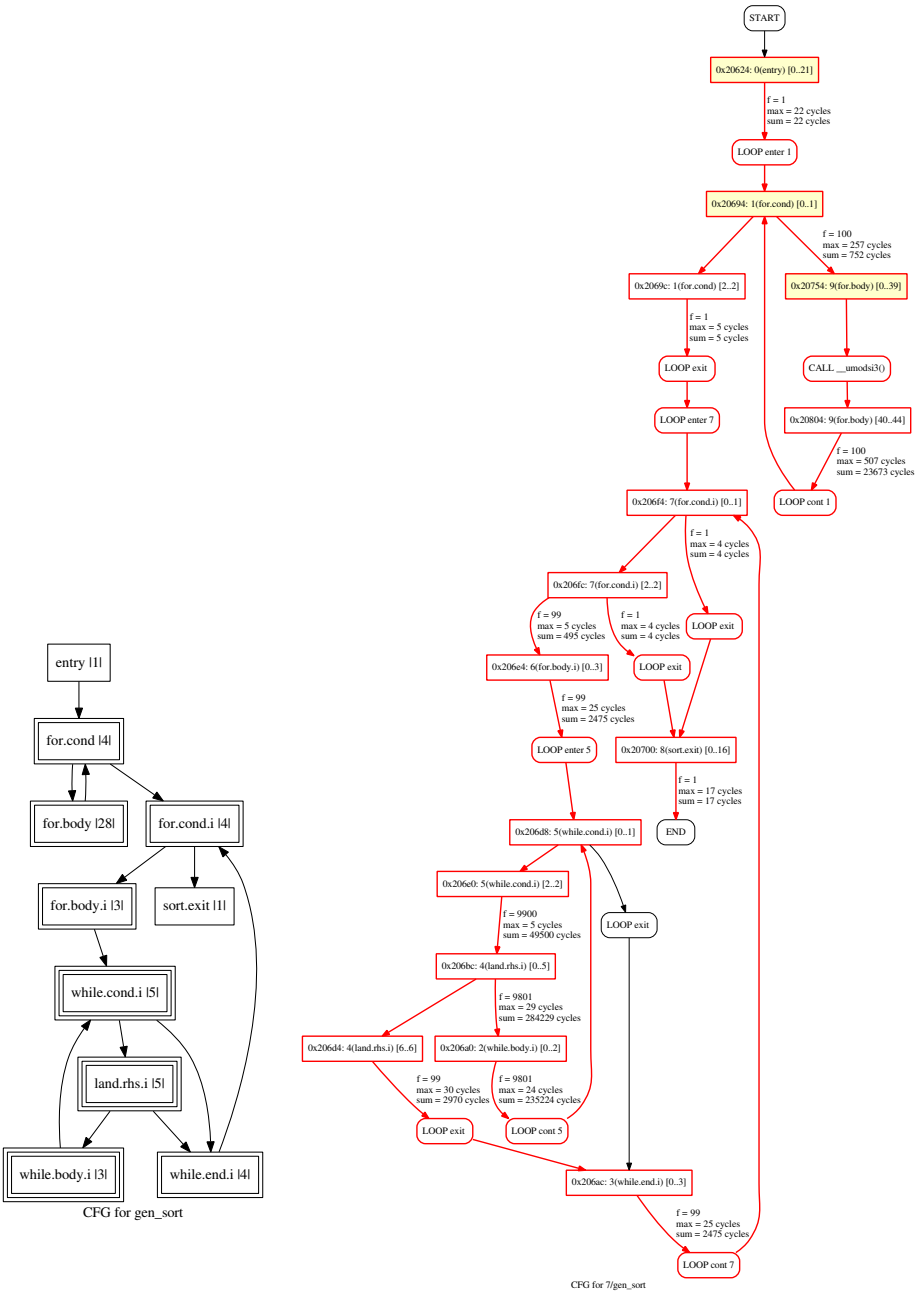
```
platin visualize -i wcet.pml -o out -f gen_sort \
                 --show-timings=platin
```

This command generates all graphs for function `gen_sort` and stores them in the output directory `out`. Figure 4 shows the generated control-flow graphs at bitcode level (after optimisation) and of the final machine code. The latter graph is the same graph that is used for WCET analysis by platin. Square boxes correspond to basic blocks or basic block slices, while round boxes are virtual nodes inserted by platin. The block node labels in the machine code graph show the address and the number of the basic block, as well as the name of the corresponding bitcode block (in brackets) and the range of the instructions in the basic block slice (in square brackets). The `--show-timings` option causes platin to highlight blocks and edges that are on the worst-case path found by the given analysis tool in the machine-code graph. Edges between basic blocks are annotated with their worst-case execution frequency and their associated WCET contribution.

6 Related Work

The WCET-aware C Compiler (WCC) [3] is a custom developed C compiler that focuses on WCET optimisation, targeting Infineon TriCore microcontrollers. It uses a machine-independent high-level intermediate representation called ICD-C for high-level optimisations, and a retargetable low-level intermediate representation called ICD-LLIR for machine optimisations and code generation. WCET analysis is performed by the AbsInt `aiT` tool at ICD-LLIR level and adds analysis results such as basic block execution times and encountered instruction cache misses, as well as information about the found worst-case path to the ICD-LLIR. The compiler maintains a mapping between the blocks of the ICD-C and ICD-LLIR representations, so that WCET analysis results can be used by high-level optimisations on ICD-C as well. Flow facts are transformed and updated by compiler and its optimisation passes itself.

Kirner et al. transform flow information in parallel to high-level optimisations such as loop interchange [8]. Their transformation technique requires control-flow update rules for optimisations that modify the control-flow graph or change loop bounds or other flow constraints. These update rules specify the relation between edge-execution frequencies before and after the optimisation, and are used to consistently transform all flow constraints affected by the optimisation. The method was implemented for source-to-source transformations but should be applicable to bitcode as well.



(a) Bitcode CFG (b) Machine-code CFG with platin WCET results

Fig. 4: Bitcode and machine-code control-flow graphs for `gen_sort`.

7 Conclusion

In this paper we presented an overview of the Patmos compiler tool chain and the platin tool kit. The platin tool kit combines several tools for compiler and WCET analysis integration, tool configuration and flow fact transformation. We demonstrated the platin tool kit on a sample application and showed how to perform a WCET analysis using platin. Due to its design, it should be possible to adapt and integrate platin into other LLVM based compilers with a low effort.

Acknowledgement

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). This paper is based on [11]. We would like to thank Christoph Cullmann and Gernot Gebhard for their work on the Patmos port of the AbsInt aiT analyser.

References

1. Abbaspour, S., Brandner, F., Schoeberl, M.: A time-predictable stack cache. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on. pp. 1–8 (June 2013)
2. Degasperi, P., Hepp, S., Puffitsch, W., Schoeberl, M.: A method cache for Patmos. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on. pp. 100–108 (June 2014)
3. Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems pp. 1–50 (2010)
4. Gomony, M.D., Garside, J., Akesson, B., Audsley, N., Goossens, K.: A generic, scalable and globally arbitrated memory tree for shared DRAM access in real-time systems. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. pp. 193–198. DATE '15, EDA Consortium, San Jose, CA, USA (2015), http://dl.acm.org/citation.cfm?id=2755753.2755795
5. Goossens, S., Kuijsten, J., Akesson, B., Goossens, K.: A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In: Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. pp. 2:1–2:10. CODES+ISSS '13, IEEE Press, Piscataway, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2555692.2555694
6. Huber, B., Prokesch, D., Puschner, P.: Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In: Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems. pp. 163–172. The Association for Computing Machinery (2013)
7. Kasapaki, E., Spars, J.: Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In: Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on. pp. 45–52 (May 2014)
8. Kirner, R., Puschner, P., Prantl, A.: Transforming flow information during code optimization for timing analysis. Real-Time Systems 45, 72–105 (2010)

9. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO'04). pp. 75–88. IEEE Computer Society (2004)
10. Lisper, B.: Sweet a tool for wcet flow analysis (extended abstract). In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, Lecture Notes in Computer Science, vol. 8803, pp. 482–485. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-662-45231-8_38
11. Puschner, P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and WCET-analysis integration. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on. pp. 1–8 (June 2013)
12. Schoeberl, M., Brandner, F., Hepp, S., Puffitsch, W., Prokesch, D.: Patmos reference handbook. Tech. rep. (2015), http://patmos.compute.dtu.dk/patmos_handbook.pdf
13. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: Towards a time-predictable dual-issue microprocessor: The Patmos approach. In: First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011). pp. 11–20 (March 2011)
14. Schoeberl, M., Silva, C., Rocha, A.: T-Crest: A Time-Predictable Multi-Core Platform For Aerospace Applications. ESA - SP, European Space Agency, ESA (2014)