The T-CREST Approach of Compiler and WCET-Analysis Integration

Peter Puschner
Daniel Prokesch
Benedikt Huber
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
{peter,daniel,benedikt}@vmars.tuwien.ac.at

Jens Knoop
Stefan Hepp
Institute of Computer Languages
Vienna University of Technology
Vienna, Austria
{knoop,hepp}@complang.tuwien.ac.at

Gernot Gebhard
AbsInt Angewandte Informatik GmbH
Saarbrücken, Germany
gebhard@absint.com

***Abstract*—A good worst-case performance and the availability of high-quality bounds on the worst-case execution time (WCET) of tasks are central for the construction of hard real-time computer systems for safety-critical applications. Timing-predictability of the whole software/hardware system is a necessary prerequisite to achieve this. We show that a predictable architecture and the tight and seamless integration of compilation and WCET analysis is beneficial to achieve the initial two goals of good worst-case performance and the availability of high-quality bounds on the WCET of computation tasks. Information generated by the compiler improves the WCET analysis. Detailed timing feedback from the WCET analysis helps the compiler to reduce the worst case execution time. The paper describes the interface and the interaction between the industrial strength WCET analysis tool and the compiler as developed in the EU FP7 T-CREST project, and demonstrates the cooperation of these tools with an illustrative example.**

I. 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. Despite the stringent timing requirements of these time-critical applications, and despite the importance to design and implement systems to meet these timing constraints and to show that the implementation indeed fulfils all timing requirements, 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.

When it comes to planning and evaluating the worst-case timing of tasks for time-critical embedded applications, the inadequacy of platforms and methods/tools leads to a number of problems questioning the precision and even soundness of the analysis results and thus to a waste of efforts.

Lack of Independence of Hardware State and Software Context: In many contemporary architectures, the time needed for executing instructions, accessing data, and using the processor's resources depends on the state of the hardware, which has built up during the history of earlier operations and events, thus creating far-reaching timing effects. These far-reaching timing effects increase the price for a high-quality timing analysis and lead to pessimism of WCET estimates.

Lack of Temporal Composability and Compositionality: Non-local, far-reaching timing effects lead to context-dependent timing which prohibits the isolated assessment of the timing of software. This impedes the modular construction and reuse of software, which builds on timing composability.

Lack of Small Timing Variability: The timing of hardware operations and software functions, as well as the control flow of software often depend on program inputs. These timing variations lead to timing behaviour that is difficult to analyse and lead to pessimism in the WCET analysis.

Lack of Seamless Integration of Compiler and WCET analyser: WCET analysis needs exact information about instructions and data being accessed by the code under analysis. Therefore it is done on the executable code, i.e., after the code has been compiled. During compilation, a lot of information about the original semantics and behaviour of the source code, including information that would be of use to the WCET analysis, is removed by the compiler. The resulting loss of information contributes to pessimism in the WCET analysis.

Lack of WCET-aware/sensitive Optimisations: Code optimisations done by traditional compilers aim at improving performance, i.e., reducing the average execution time. These optimisations are often detrimental for real-time systems, as they can increase the WCET of the code. Further, a number of optimisations cannot be reconstructed from the object code or executable code, thus (a) again making the WCET analysis overly pessimistic and (b) prohibiting a meaningful back annotation of WCET-timing information from the machine-code to the source-code level.

Within the T-CREST project we aim at making the quest for time-predictability a guiding principle in the design of computing platforms and code for time-critical embedded systems. To this end we are developing a novel time-predictable processor architecture [1] and a compiler that interacts with WCET-analysis to produce code that has stable execution times, is highly competitive w.r.t. its worst-case performance, and is well amenable to a high-quality WCET analysis.

In this paper we present the T-CREST approach to an integrated compilation and timing analysis. This tight integration of compiler and the WCET tool allows us to access and connect both (a) information about the structure and semantics of the code being compiled and (b) detailed worst-case timing information about all code parts represented by the data structures in the compiler in a way that has not been possible before. As a result of this tool integration we are able to produce time-predictable/WCET-analysable code for which all the worst-case timing details are transparent and to make optimisations that yield a good WCET-performance. The paper highlights the T-CREST goals (Section II), describes the WCET tool (Section III), the T-CREST compiler (Section IV), the integration of compiler and WCET analysis (Section V), and demonstrates the cooperation of compiler and WCET analysis by means of an example (Section VI).

II. T-CREST GOALS

The central goal of T-CREST is to develop a system architecture and a tool framework that regard the timing comprehensibility of all hardware and software activities as a primary objective in hardware and software development. A detailed understanding of the timing of all components of a system from the very beginning of its construction is a basic prerequisite for building time-predictable embedded computer systems for time- and safety-critical applications.

To achieve this overall goal of T-CREST we strive for the following goals for the compiler and WCET analysis.

Temporal Composability: The execution times of generated code should not depend on the software context in which the code is executed, i.e., changing one part of some software must not change the timing of the other code sections. Temporal composability is a prerequisite for a hierarchical, modular software development and the time-aware re-use of software.

Timing Compositionality: Given the timing of pieces of code, the timing of a composite should be easy to analyse, i.e., computable with simple timing formulae. Timing compositionality helps to keep the overestimation of the worst-case execution time and the resource needs for WCET analysis low.

Small Timing Variability: Ideally, the generated code should run with constant execution time or at least with small execution-time jitter (variability). This greatly simplifies both timing analysis and the argumentation about the temporal behaviour of real-time application software.

Good Worst-Case Performance: In hard real-time applications, the average execution time (performance) of code is of minor concern. The generated code should have a short WCET. A short WCET together with a tight WCET estimate helps to keep the resource needs of an application low, thus reducing the prime cost of the computer system.

In order to reach these goals, the compiler and WCET analysis have to rely on the availability of an appropriate hardware platform (see Section IV-A). The rest of the paper will explicate that the listed goals can only be reached through a close interaction between the compiler and the WCET tool.

III. WCET ANALYSIS

A. WCET Analysis Framework

A static worst case execution time analyser typically works in four phases as illustrated in Figure 1:

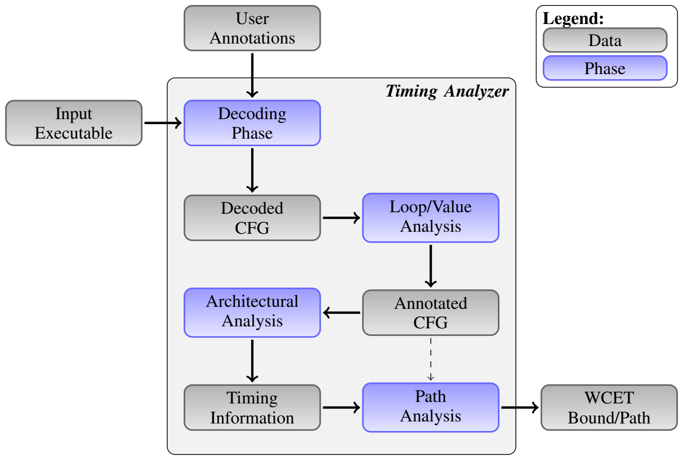


Fig. 1. Components of a WCET analysis framework.

Decoding Phase: In this phase the tool processes the input (binary) program and an annotation file that provides further knowledge (see Section III-B). The decoder identifies the machine instructions and reconstructs the control-flow graph. The user may provide additional information that is passed to each analysis phase. Such information could be targets of computed calls (used during the decoding phase), the number of iterations for a specific loop (used during the loop and value analysis), the hardware configuration (as required by the pipeline analysis), and flow constraints (used in path analysis).

Loop and Value Analysis: The loop analysis phase tries to automatically compute upper bounds of loop iterations for all loops. The user may refine computed loop bounds and specify bounds that could not be computed automatically.

The value analysis determines safe approximations of the values of processor registers and memory cells for every program point and execution context. Contents of registers or memory cells as well as address ranges for memory accesses may be provided by user annotations.

Architectural Analysis: Processing the annotated control-flow graph, the architectural analysis simulates the execution behaviour of the input program through an abstract hardware model. The analysis determines lower and upper bounds for the execution times of basic blocks by performing an abstract interpretation of the program execution on the particular architecture, taking into account its pipeline, caches, memory buses, and attached peripheral devices [2]–[4].

Typically, the architectural analysis is a composition of a pipeline analysis and a cache analysis. By means of an abstract model of the hardware architecture, the pipeline analysis simulates the execution of each instruction. The cache analysis provides safe approximations of the contents of the caches at each program point. Complex architectural features are the main challenges for this analysis phase.

Since most parts of the pipeline state influence timing, current abstract models closely resemble the concrete hardware. The more performance-enhancing features a pipeline has, the larger is the search space. Superscalar- and out-of-order execution increase the number of possible interleavings. The larger the buffers (fetch buffers, retirement queues, etc.), the longer the influence of past events lasts. Dynamic branch prediction, speculative execution, cache-like structures, and branch history tables increase history dependence even more.

All these features influence execution time. To compute a precise bound on the execution time of a basic block, the analysis needs to exclude as many *timing accidents* as possible. Such accidents are data hazards, branch mis-predictions, occupied functional units, full queues, etc.

Abstract states may lack information about the state of some processor components, e.g., caches, queues, or predictors. Transitions of the pipeline may depend on such missing information. Thus the abstract pipeline model becomes non-deterministic even though the concrete pipeline is deterministic. When dealing with this non-determinism, one could be tempted to design a WCET analysis such that only the locally most expensive pipeline transition is chosen. However, in the presence of timing anomalies [5], [6] this is unsound. Thus, the analysis has to follow all possible successor states.

Path Analysis: Using the results of the preceding loop and value analysis and the pipeline analysis phases, the path analysis estimates the worst-case path and computes a safe WCET estimate. The analysis translates the control-flow graph with the basic block timing bounds determined by the pipeline analysis and the loop (and recursion) bounds derived by the loop and value analysis into an integer linear program [7]. The solution of the ILP yields the worst-case path and a safe WCET approximation.

The WCET analyser `aiT` implements this architecture, *cf.* http://www.absint.de/ait, and has been shown to determine precise execution-time bounds [4], [8]–[10]. `aiT` has been used in the aeronautics and automotive industries.

B. AIS Annotations

Apart from the executable, `aiT` needs user input to find a result at all, or to improve the precision of the result. The most important user annotations specify the targets of computed calls and branches and the maximum iteration counts of loops (there are many other possible annotations).

Targets of Computed Calls and Branches: For a correct reconstruction of the control flow from the binary, targets of computed calls and branches must be known. `aiT` can find many of these targets automatically for code compiled from C. This is done by identifying and interpreting switch tables and static arrays of function pointers. Yet dynamic use of function pointers cannot be tracked by `aiT`, and hand-written assembler code in library functions often contains difficult computed branches. Targets for computed calls and branches that are not found by `aiT` must be specified by the user. This can be done by writing specifications of the following forms in a parameter file called AIS file.

Example: The library routine `C_MEMCPY` in TI's standard library for the TMS470 consists of hand-written assembler code. It contains two computed branches whose targets can be specified as follows:

```
instruction "C_MEMCPY" + 1 computed
branches to pc + 0x14 bytes, pc + 0x24 bytes;

instruction "C_MEMCPY" + 2 computed
branches to pc + 0x10 bytes, pc + 0x20 bytes;
```

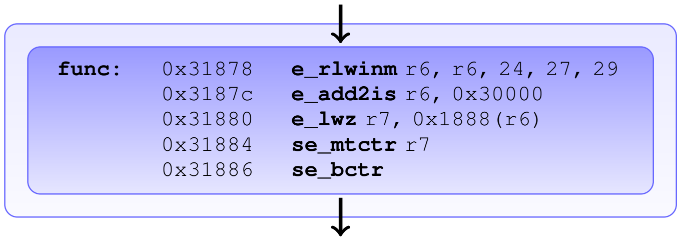
Loop Bounds: WCET analysis requires that upper bounds for the iteration numbers of all loops be known. `aiT` tries to determine the number of loop iterations by *loop bound analysis*, but succeeds in doing so only for loops with constant bounds whose code matches certain patterns typically generated by the supported compilers. Bounds for the iteration numbers of the remaining loops must be provided by user annotations.

Example: The following annotation specifies that the first loop in `prime` has the loop test at the end and is executed at most 10 times:

```
loop "prime" + 1 loop end max 10;
```

C. Motivating Example

Bernat and Holsti [11] proposed a "wish list" regarding the information a compiler should provide to a WCET analysis tool chain. Some of this information is already made available. For example, the compiler can generate line number information that provides a mapping of source code to object code. However, the compiler omits most of the information that can be useful for timing analysis.

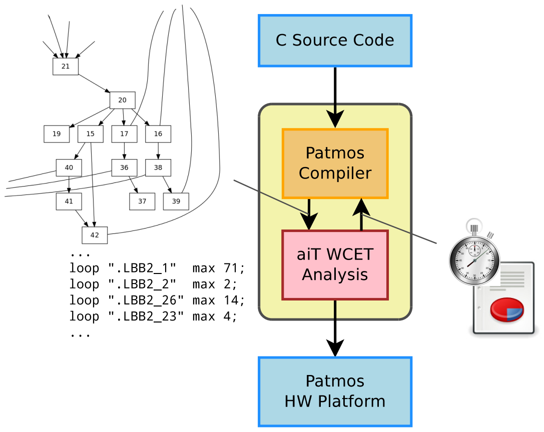


```
func:    0x31878    e_rlwinm r6, r6, 24, 27, 29
         0x3187c    e_add2is r6, 0x30000
         0x31880    e_lwz r7, 0x1888(r6)
         0x31884    se_mtctr r7
         0x31886    se_bctr
```

Fig. 2. Compiler-generated switch table code that lacks a size check on the index register `r6` (DiabData v5.8 compiler for PowerPC).

Consider the code snippet of the function `func` depicted in Figure 2, which represents the PowerPC object code for a switch table. The register `r6` contains the value of the switch table index. The first instruction at `0x31878` left-rotates the register value by 24 bits and masks out all bits except for the least significant bits 3, 4 and 5. Hence, the register `r6` can accept eight different values (i.e., $\texttt{r6} = 2^2 i$ for $0 \leq i < 8$). The next instruction adds the constant `0x30000` to `r6`. The `e_lwz` instruction reads a 32bit word from read-only memory at $\texttt{0x31888} + \texttt{r6}$. The remaining instructions move the value read from memory into the `ctr` register and branch to `ctr`.

Usually, the compiler generates a switch table size check to branch to the *default* case if the switch expression (i.e., the index) exceeds the switch table range. In the above example, the compiler has omitted the size check. Without further user annotations `aiT` cannot safely determine how many cases are being addressed by the code snippet. A naive assumption would be that eight targets are possible because `r6` can only accept eight different values. However, the compiler could have known statically that some index values are not feasible at all, which would render a reconstruction of the program control-flow based on this assumption invalid.



```
...
loop ".LBB2_1"  max 71;
loop ".LBB2_2"  max 2;
loop ".LBB2_26" max 14;
loop ".LBB2_23" max 4;
...
```

Fig. 3. The compiler should preserve information about program structure and flow constraints, while let optimisations be guided by timing analysis results.

At compile time the real switch table size is known. Hence, the compiler can provide all feasible control-flow successors in form of intermediate annotations to WCET analysis. Providing this information ensures a correct control-flow reconstruction.

There are other scenarios where the compiler can provide information to aid WCET analysis. Consider, e.g., a loop where the compiler unrolls the first four loop iterations. If the loop analysis is unable to infer a loop bound automatically the user has to provide a safe upper bound. The user is usually unaware of code optimisations, such as loop unrolling. Again, if the compiler provided that knowledge, a WCET analysis would certainly be able to compute tighter timing bounds.

IV. Compiler

Prior to WCET analysis the compiler needs to translate a program specification written in a high level programming language to an executable for the hardware platform.

Within T-Crest, we develop a processor from scratch that is designed for high time predictability [1]. The architectural features of this processor, which is named *Patmos*, are carefully designed to improve performance yet remain inherently timing analysable. An integral part of the design philosophy is to use static (compile-time) alternatives for commonly used performance-enhancing features at runtime, to reduce the dynamic behaviour and state of the processor that is not visible or cannot be controlled by the code at the instruction set architectural (ISA) level. In conformance with this philosophy, the compiler must generate code that targets the Patmos ISA and exerts control over these components.

Besides the hardware architectural means to obtain tight WCET bounds, we pursue a tight integration of the compilation process and timing analysis, as illustrated in Figure 3. On one hand, the compiler preserves information available during the compilation process that usually is discarded but otherwise would be 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*). On the other hand, results from timing analysis are fed back to the compiler, to guide optimisations that aim at reducing the guaranteed worst-case performance.

A. Support for the Patmos Time-Predictable Processor

The architectural design for time predictability requires dedicated support from the compiler.

The processor features a predictable, statically scheduled dual-issue RISC pipeline with a fully predicated instruction set. The absence of dynamic instruction reordering and assignment to the available functional units requires the compiler to create a feasible instruction schedule, including respecting instruction latencies, bundling instructions for dual-issue and filling delay slots of control-flow instructions. Predicated execution is a requirement to support single-path code generation [12], [13]. Single-path code aims at reducing the execution time variability of tasks caused by input-data dependent control-flow decisions.

In terms of local memory, Patmos provides a set of specialised caches and a data scratchpad, which are optimised for different uses [14]. The reduced interference between the memories potentially simplifies both hardware design and cache analysis.

The *method cache* is an instruction cache that caches whole functions, and allows for large continuous block-transfers, but is more dynamic than a conventional instruction scratchpad. It does not interfere with any data caches and cache misses and cache updates only occur at a very limited number of instructions. The compiler splits functions that otherwise are too large to fit in the method cache as a whole, or for optimisation purposes.

The *stack cache* is an explicitly managed local data cache for the call stack. Similar to the method cache, spilling and filling the stack cache to main memory is performed using efficient block transfers and can only occur at special control instructions the compiler needs to insert, typically at function entries, calls and returns. The compiler can allocate data objects to the stack frame for which it can guarantee that the stack frame is not evicted while they are live (e.g. register spill slots, local variables with fixed size). The stack cache also does not interfere with other caches.

Additionally to a set-associative *data cache* with LRU replacement policy and write-through write policy to facilitate precise cache analysis, Patmos provides a local *data scratchpad memory* with very low and constant access time.

The Patmos ISA exposes the type of physical memory accessed in *typed memory instructions*. Hence, the compiler generates different instructions for accessing the main memory through the data cache, or for accessing data on the local stack cache or scratchpad memory. Patmos provides special instructions to bypass the data cache in order to retain the cache state in cases where storing data in the cache is not beneficial due to the lack of spatial or temporal locality, and is a means to obtain more precise cache analysis results in the architectural analysis phase, *cf.* Section III-A.

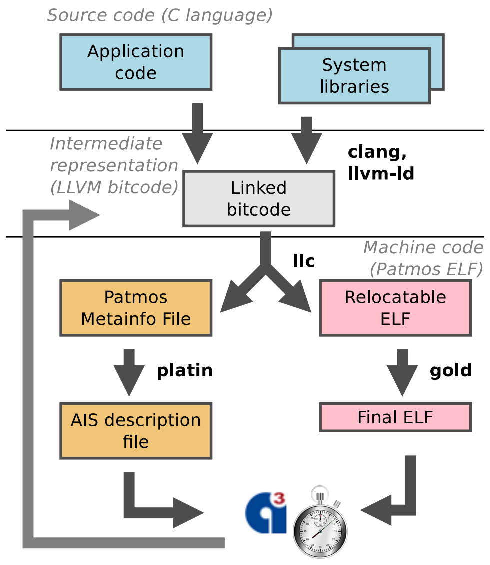


Fig. 4. Compiler Tool Chain Overview

B. *Compilation Tool Chain*

Figure 4 gives an overview of the compiler tool chain. The compiler is based on the LLVM compiler framework [15]. 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 and static standard and support libraries are linked on this intermediate level by the `llvm-ld` tool, presenting subsequent analysis and optimisation passes as well as the code generation backend a complete view of the whole program. This control-flow graph oriented intermediate representation is particularly suitable for generic target independent optimisations, such as common subexpression elimination, which are readily available through `llvm-ld`. The `llc` tool constitutes the backend translating LLVM bitcode into machine code for the Patmos ISA, addressing the target-specific features for time predictability. In addition to the machine code, the backend exports complete information about the control-flow structure on both bitcode and machine code, a relation between these two representations, as detailed in Section V. The backend produces a relocatable ELF binary containing symbolic address information, which is processed by `gold`,[1] defining the final data and memory layout, and resolving symbol relocations. 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 view of the program that is necessary to optimise the WCET.

V. Integration of Compiler and WCET Analysis

Bernat and Holsti [11] identify four categories of features the compiler should provide to support WCET integration: providing information about the source code's semantics, relating source and machine code, passing information about machine code to the compiler, and control over performed optimisations and machine code generation. As the WCET analysis tool carries out its analysis on binary code, we interpret the first two categories as the challenge to translate information from source code or intermediate code to machine code. This information includes e.g. points-to sets, targets of indirect calls or additional flow information, and stems from both analysis tools operating on higher-level representations and user annotations. Approximate relations between machine code and source code are made available by existing compilers. While this information is useful for providing feedback to humans, it is not suitable for performing sound WCET analysis.

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

We do not only provide the compiler's knowledge to the WCET analysis tool, but also want to use WCET analysis results to direct optimisations. The WCET analysis tool should thus provide the WCET of program fragments (functions, loops, basic blocks), as well as information on the worst-case path and WCET-criticalities [16].

A. *Compilation Flow and Preservation of Metainformation*

Due to the complexity of modern compilers and their optimisations, transforming information from the source level to the machine-code level is not trivial to retrofit into an existing industrial-quality framework. In order to manage the complexity of this problem, we subdivide the transformation into a number of steps. The compilation flow described in Section IV permits to use different strategies for the preservation of meta-information in different stages of compilation. We outline this strategy below.

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

High-Level Optimisations: High-level optimisations are performed on bitcode, although the same considerations apply to source-to-source optimisations. Some of the available high-level optimisations perform major structural changes to the program (e.g., loop unswitching). Consequently, these optimisations need to be extended to preserve meta-information which is relevant for timing analysis but not necessarily for the compiler backend. LLVM partly addresses this problem, as it provides infrastructure for bitcode meta-information, and helps to transform or invalidate debug information during optimisations. Techniques for maintaining for example loop bounds, which are crucial for WCET analysis, have been developed [17], but require considerable additional effort for each optimisation. However, as these optimisations are implemented in a platform-independent way, it is likely that the investments on preserving meta-information pay off.

From Bitcode to Machine Code: In order to preserve meta-information in the compiler backend, the compiler maintains relations between basic blocks (and memory accesses) at

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

the bitcode and the machine code level. Flow facts are transformed to machine code using these relations. Thus it is not necessary to add dedicated support for flow-fact updates in the backend. Those high-level optimisations that perform structural changes that cannot be expressed in our relation model, need to be (and are) implemented as bitcode optimisations.

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

B. Information Exchange Language and Tools

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

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

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

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

C. Current State

The integration of compiler and WCET analysis is under active development. At the current implementation stage, support for emitting the control-flow structure of both the bitcode and machine code, as well as the relation between them is present in `llc`. The resulting PML database is then subsequently manipulated by the tools of the `platin` toolset.

In addition to routines for merging information from different PML files to improve modularity, and a visualisation tool to present control-flow information to humans, functionality to interact with the timing analyser is implemented. In particular, `platin` includes a tool to export an AIS annotation file from a PML database, and a tool to parse the analysis results and merge them into the PML database.

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

Support for user-provided flow annotations above the bitcode intermediate level is not yet implemented in the compiler. It involves extending the `clang` frontend to correctly translate flow annotations (most importantly loop bounds) in the form of C pragmas from the AST-oriented representation to the CFG-oriented bitcode representation, and extending selected relevant CFG-manipulating transformations at bitcode level (e.g., loop unswitching) to co-transform the flow annotations; both of which we plan to implement in the near future.

VI. Example

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

Base64 is an encoding scheme to represent binary data in an ASCII string format by translating it into a radix-64 representation. Therefore, Base64 encoding converts 3 octets into 4 encoded characters. Figure 5 depicts the C source code of a simplified version of the decoding function. Depending on the current position within four read characters, binary data is written with different bit offsets to the target buffer. This behaviour is realised with a state-machine implemented as switch statement.

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

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

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

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

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

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

Fig. 5. Base64 decoding function. The switch statement with unreachable default branch is translated to a jump table.

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

Fig. 6. The assembly generated for `b64_pton` contains an indirect branch (marked with ∗). No switch table size check is performed.

Lacking an implementation thereof at the current state, we use the opportunity to demonstrate the benefit of the integration of the Patmos simulator `pasim` into the `platin` toolset, as illustrated in Figure 8: We simulate the decoding function with a string of maximally allowed length as input data and let the trace-analysis tool extract the loop iteration counts for the executed loops, which are annotated back to the PML database. Finally, an AIS annotation file is exported from the latter, containing the relevant annotations enabling WCET-analysis, as shown in Figure 9.

VII. Related Work

The WCET-aware C Compiler (WCC) [18] 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

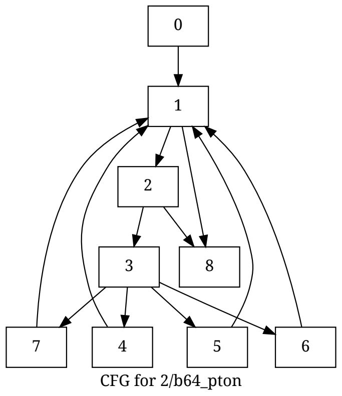


Fig. 7. Machine-code level control-flow graph of the `b64_pton` function from Figure 5.

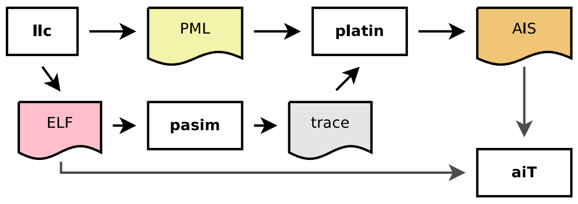


Fig. 8. Flow diagram when using `platin` to extract loop bounds of a simulation trace.

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.

The Java Optimized Processor (JOP) [19] implements the Java Virtual Machine (JVM) in hardware and is designed to simplify WCET analysis. The processor executes a RISC-style microcode instruction set, with well-defined timing behaviour. An additional pipeline stage translates JVM instructions to a sequence of microcode instructions, in a predictable way. As the processor executes Java bytecode, there is no traditional compiler backend in the JOP tool chain. The bytecode instructions executed by JOP provide information about the static types of variables, dynamic dispatch targets and accessed memory areas, solving some of the integration challenges by design. Optimisations are performed on bytecode and high-level optimisations (inlining, loop optimisations) preserve user-provided information, such as loop bounds [20].

Precision Timed Machines (PRET) [21], [22] are a computer architecture designed for repeatable and predictable performance. Characteristic features are the thread-interleaved pipeline, an exposed memory hierarchy and timing instructions, exposing timing behaviour to the ISA. Most notably, a deadline instruction sets the execution-time limit for subsequent instructions until the next deadline instruction is encountered, which then stalls the execution until the specified time has elapsed. A code generator could leverage this instruction to create applications with repeatable timing, i.e., low execution-

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

Fig. 9. The AIS annotations exported from the PML database for the b64_pton function.

time variability. The absence of implicit hardware state and dependencies between different hardware threads aims at making the timing behaviour composable and easily analysable.

Kirner et al. transform flow information in parallel to high-level optimisations such as loop interchange [17]. 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.

VIII. Conclusion

The T-CREST project aims at constructing a time-predictable multi-core platform for embedded systems. This platform comprises both the Patmos processor architecture and an integrated compilation and WCET-analysis framework that specifically targets at generating time-predictable code, optimised in order to keep the WCET of the code short.

To achieve time-predictability and WCET optimisation the T-CREST compiler and the aiT tool maintain and exchange information about the code's structure, possible execution paths, and code timing. The information exchange is supported by a number of tools and routines that process and manage timing-relevant information about the code at all code-representation levels of the compilation process, without a gap. A specific information-exchange language and file format, called PML, is used to support the seamless tool interaction.

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).

References

[1] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, March 2011, pp. 11–20.

[2] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*. London, UK: Springer-Verlag, 2002, pp. 334–348.

[3] S. Thesing, "Safe and precise WCET determinations by abstract interpretation of pipeline models," Ph.D. dissertation, Saarland University, 2004.

[4] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[5] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Real-Time Systems Symposium (RTSS)*, December 1999.

[6] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Workshop on Worst-Case Execution-Time Analysis (WCET)*, July 2006.

[7] H. Theiling, "ILP-based interprocedural path analysis," in *EMSOFT*, 2002, pp. 349–363.

[8] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Conference on Embedded Software (EMSOFT)*, ser. LNCS, vol. 2211, 2001.

[9] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand, "An abstract-interpretation-based timing validation of hard real-time avionics software systems," in *Dependable Systems and Networks (DSN)*, June 2003.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Real-Time Systems*, vol. 91, no. 7, pp. 1038–1054, 2003.

[11] G. Bernat and N. Holsti, "Compiler support for WCET analysis: a wish list," in *WCET*, 2003, pp. 65–69.

[12] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for time predictability," in *Proc. SAFECOMP 2012 Workshops (LNCS 7613)*. Springer, 2012, pp. 382–391.

[13] P. Puschner and A. Burns, "Writing temporally predictable code," in *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2002, pp. 85–91.

[14] M. Schoeberl, "Time-predictable cache organization," in *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*, ser. STFSSD '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 11–16.

[15] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, 2004, pp. 75–88.

[16] F. Brandner, S. Hepp, and A. Jordan, "Static profiling of the worst-case in real-time programs," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12. New York, NY, USA: ACM, 2012, pp. 101–110.

[17] R. Kirner, P. Puschner, and A. Prantl, "Transforming flow information during code optimization for timing analysis," *Real-Time Systems*, vol. 45, pp. 72–105, 2010.

[18] H. Falk and P. Lokuciejewski, "A compiler framework for the reduction of worst-case execution times," *Real-Time Systems*, pp. 1–50, 2010.

[19] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a java processor," *Softw. Pract. Exper.*, vol. 40, no. 6, pp. 507–542, May 2010.

[20] S. Hepp and M. Schoeberl, "Worst-case execution time based optimization of real-time java programs," *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 64–70, 2012.

[21] S. A. Edwards and E. A. Lee, "The case for the precision timed (pret) machine," in *Proceedings of the 44th annual conference on Design automation*. SESSION: Wild and crazy ideas (WACI), June 2007, pp. 264 – 265.

[22] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*, October 2012.