The Heptane Static Worst-Case Execution Time Estimation Tool

Damien Hardy[1]**, Benjamin Rouxel**[2]**, and Isabelle Puaut**[3]

1 **University of Rennes 1, Rennes, France**
`damien.hardy@irisa.fr`
2 **University of Rennes 1, Rennes, France**
`benjamin.rouxel@irisa.fr`
3 **University of Rennes 1, Rennes, France**
`isabelle.puaut@irisa.fr`

Abstract

Estimation of worst-case execution times (WCETs) is required to validate the temporal behavior of hard real time systems. Heptane is an open-source software program that estimates upper bounds of execution times on MIPS and ARM v7 architectures, offered to the WCET estimation community to experiment new WCET estimation techniques. The software architecture of Heptane was designed to be as modular and extensible as possible to facilitate the integration of new approaches. This paper is devoted to a description of Heptane, and includes information on the analyses it implements, how to use it and extend it.

1998 ACM Subject Classification C.3 Real-time and embedded systems

Keywords and phrases Worst-Case Execution Time Estimation, Static Analysis, WCET Estimation Tool, Implicit Path Enumeration Technique

Digital Object Identifier 10.4230/OASIcs.WCET.2017.8

1 Introduction

Knowing task worst-case execution times (WCET) is of prime importance for the timing analysis of hard real-time systems. Timing analysis of multi-task software is in general made of two levels: *WCET analysis* and *schedulability analysis*. WCET analysis estimates the worst-case timing requirements of an isolated task. At this level, activities other than ones related to the considered task (interrupts, blocking, pre-emptions or any kind of interference from other tasks in the system) are ignored. At the *schedulability analysis* level, the analysis considers multiple tasks executing on the processor and competing for resources, and thus may block while attempting to access the resources.

WCETs may be obtained using *static analysis* techniques, *measurement-based* techniques or hybrid techniques (see [29] for a survey). A static WCET analysis tool provides an upper bound (WCET estimate) on the time required to execute a given code on a given hardware without program execution. A static WCET analysis tool should be able to work at a high level to determine the longest path in a code. It should also work at low-level (hardware-level), to capture the worst impact of the target processor on timing. Static WCET analysis is complicated due to the presence of architectural features that improve the processor performance: instruction and/or data caches, branch prediction and pipelines for example. Precisely modeling these architectural features is the key to have precise WCET estimates.

A number of static WCET analysis tools exist. The objective of this paper is to give a high-level and up-to-date view of the static WCET analysis tool we have designed and maintained over the years, named Heptane[1]. The first version of Heptane was developed in the late nineties during the PhD thesis of Antoine Colin. That first version, described in [6] was a research prototype written in OCaml, and originally implemented *tree-based* WCET calculation. At that time, it included the analyses required to obtain WCETs for a Pentium 1 processor (cache analysis, pipeline analysis, branch prediction analysis). Heptane was re-developed in 2003 to have a cleaner software architecture, use C++ instead of OCaml, and support more target processors. This second version [29] implemented both *tree-based* and IPET calculation. This version was used up to 2010 to implement all research related to WCET estimation in our group (compiler-directed branch prediction, cache locking, scratchpad management, analysis of data caches and shared caches in multi-cores, etc.). We then decided to change our development strategy to ease code readability and development of new analyses. We selected to integrate in the main branch of the tool only a minimum number of robust analyses. At that point in time, the software architecture of Heptane was refined again based on our past experience. This paper describes the last version of Heptane.

The aim of Heptane is to produce upper bounds of the execution times of applications. It targets applications with hard real-time requirements (automotive, railway, aerospace domains). Heptane computes WCETs using static analysis at the binary code level. It includes static analyses of micro-architectural elements such as caches and cache hierarchies.

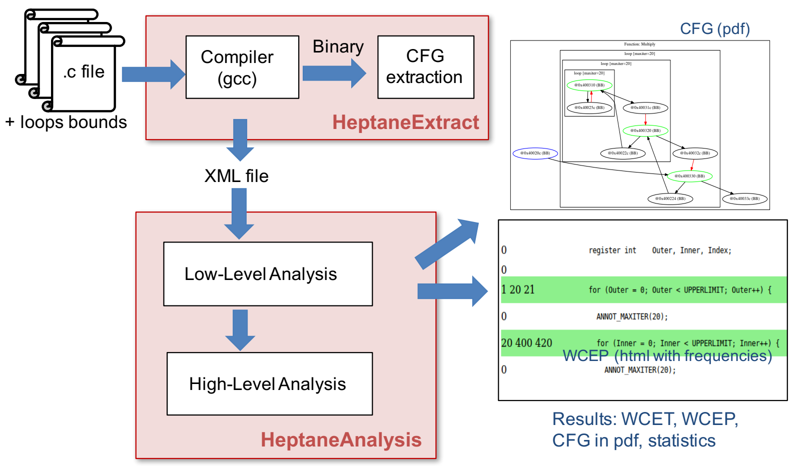
Heptane is an open source software program available under GNU General Public License v3[2]. Heptane is now a reliable research prototype, developed in C++ (approximately 13,000 lines of code) and supports MIPS and ARM v7 instruction sets. In particular, we are using a continuous integration framework to check that the tool builds correctly and perform non regression testing, for the supported target processors and host operating systems. Heptane was demonstrated during the 1st Tutorial on Tools for Real-Time Systems [28].

As compared to other open-source static WCET analyzer tools, Heptane has a special focus on cache analysis (analysis of cache hierarchies, support for multiple replacement policies), but currently supports only two target processors. OTAWA [20] supports more processor architectures than Heptane but implements less advanced cache analyses. SWEET [26] focuses on flow fact analysis and does not include any hardware-level analysis. Bound-T [4] supports different processor architectures but does not provide a cache analysis. Platin is a static WCET estimation tool dedicated to the analysis of the Patmos architecture [24]. Chronos [5] implements advanced low-level analyses, but is limited to the SimpleScalar architecture.

The commercial tool aiT [2] is the most advanced static WCET analysis tool, but unfortunately is not open-source and as such cannot be extended for research studies that require the tool to be modified.

The remainder of the paper provides more details on Heptane and is organized as follows. An overview of Heptane is presented in Section 2. Section 3 describes the WCET estimation techniques implemented in Heptane. The usage of Heptane is detailed in Section 4. Some number to evaluate the performance of Heptane are given in Section 5. Some hints to extend Heptane are mentioned in Section 6. Finally, conclusions and plans for future extensions of the tool are given in Section 7.

[1] The web site of Heptane is: `https://team.inria.fr/pacap/software/heptane/`.
[2] Heptane is registered with APP (Agence de Protection des Programmes) under number IDDN.FR.001.510039.000.S.P.2003.000.10600.



```
0               register int    Outer, Inner, Index;
0
1 20 21         for (Outer = 0; Outer < UPPERLIMIT; Outer++) {
0                 ANNOT_MAXITER(20);
20 400 420        for (Inner = 0; Inner < UPPERLIMIT; Inner++) {
0                   ANNOT_MAXITER(20);
```

Figure 1 Heptane toolchain: from source code to WCET estimate.

2 Overview of Heptane

2.1 The Heptane toolchain

As illustrated in Figure 1, Heptane is divided in two parts: *HeptaneExtract* for Control Flow Graph (CFG) extraction and *HeptaneAnalysis* for the actual WCET estimation.

HeptaneExtract generates a control flow graph (CFG) from a set of program source-code files written in C or assembly language. It calls the compiler and linker to generate the binary file and then construct the CFG. After the construction of the CFG, *HeptaneExtract* identifies the different loops, attaches the loop bounds information provided in the source file and attaches the instruction addresses based on the binary file. The CFG is stored in XML format to be used by *HeptaneAnalysis* or be manually inspected if needed.

HeptaneAnalysis applies low-level and high-level analyses to produce the WCET estimate. *HeptaneAnalysis* can also be used to generate extra information like a graphical representation of the CFG, a HTML version of the source that highlights the lines on (one of) the longest execution path(s) and some statistics of the application under study.

The CFG produced by *HeptaneExtract* will be gradually enriched by every analysis of *HeptaneAnalysis* with the analysis results. For that purpose, a library named *cfglib* has been developed. This library is designed to manage an extensible program representation that includes objects for programs, functions, loops, basic blocks, edges, instruction, etc. The library also provides so-called *attributes* that can be attached to any of the objects that represent the program. The library provides a small number of built-in attribute types (string, integers, floats, etc.), and new attribute types can be defined for the purpose of an analysis. When attaching an attribute to an object (e.g. a loop), an analysis developer calls the attribute attachment function of *cfglib* for the corresponding attribute type (e.g. integer) and provides as inputs the name of the attribute to be attached (e.g. “maxiter”) and the corresponding value (e.g. 10). The library includes serialization and deserialization facilities for objects and built-in attribute types. As explained later, each analysis may export its results in XML.

2.2 Source code constraints

Since Heptane does not include any analysis of maximum numbers of loop iterations, the source code has to be augmented by the user with annotations to provide loop bounds. At the beginning of each loop body, a dedicated macro named ANNOT_MAXITER has to be inserted (see Listing 1). The macro is defined in the header *"annot.h"* that has to be included. Loop bounds are *local* loop bounds (maximum number of iterations for each entry in the loop) and should be constant values. The macro expands to assembly code, that will create a specific section in the final binary with the maximum number of iterations.

```
#include "annot.h"
int i,j;
for (i = 0; i < 20; i++) {
     ANNOT_MAXITER(20);
     for (j = 0; j < 10; j++) {
          ANNOT_MAXITER(10);
          ...
          }
}
```

Listing 1 ANNOT_MAXITER usage.

Furthermore, some restrictions on the C code are required to be able to generate the CFG statically:

- Pointers to functions are not supported to be able to generate the call graph.
- Indirect jumps and jump tables are not supported. They can be generated for instance by switches constructs.
- Each loop must have a single entry. The identification of loops in Heptane uses DJ Graphs [25]
- Pointer arithmetic is not supported by the data address analysis

Programs not meeting these restrictions are detected and reported as errors. Finally, to be able to perform a correct matching between the loop bounds defined in the source file and the binary file the use of compiler optimization should not affect this matching. It is the user responsibility to select the compiler optimizations that do not change loop bounds.

3 WCET estimation in Heptane

Static WCET estimation methods are generally divided into two steps, commonly named *high-level* analysis and *low-level* analysis. The high-level analysis determines the longest execution path among all possible flows in a program. The low-level analysis is used to account for the processor microarchitecture. In *HeptaneAnalysis*, each analysis is *contextual*, meaning the analysis of a function is performed for every call path of the function (*e.g.* $main \rightarrow g \rightarrow f$ and $main \rightarrow f$ for a function f called directly by function *main* and also called by g that is called by *main*).

3.1 High-level analysis

For the *high-level* analysis, *HeptaneAnalysis* implements the most prevalent technique, named IPET for *Implicit Path Enumeration Technique*[17]. IPET is based on an Integer Linear Programming (ILP) formulation of the WCET calculation problem. It reflects the program structure and the possible execution flows using a set of linear constraints. An upper bound of the program's WCET is obtained by maximizing objective function

$\sum_{i \in BasicBlocks} T_i * f_i$. T_i (constant in the ILP problem) is the timing information of basic block i. T_i integrates the effects of micro-architecture, and is determined by the low-level analysis.

The variable f_i in the ILP system, to be instantiated by the ILP solver, corresponds to the number of times basic block i is executed. The values of all variables f_i, once set by the ILP solver to maximize the objective function, identify a set of paths in the program leading to the estimated WCET.

3.2 Low-level analysis

For the *low-level* analysis, *HeptaneAnalysis* implements a *data address analysis*, a *cache analysis* and a *pipeline analysis*.

Data address analysis

The *data address analysis* conservatively determines the addresses of referenced data. In case the exact address of the referenced data cannot be determined, a range of addresses is conservatively provided. The addresses of instructions are determined during the CFG extraction thanks to the addresses present in the binary file.

Heptane includes a *stack analysis*, that calculates the range of addresses for every stack frame, in any call context. The analysis assumes an acyclic call graph, which is common in real-time systems because recursion raises predictability issues. The analysis relies on the knowledge of the base of the stack based address, given as input to *HeptaneAnalysis*, and the size of all functions' stack frames, obtained by scanning the first instruction of every function, responsible for allocating the stack frame. The address of every stack frame is obtained by propagating the address of the stack frame for every callee of every function along the acyclic call graph.

Data address analysis is implemented by an inter-procedural data flow analysis based on abstract interpretation, that evaluates the contents of every register before and after each instruction. Since Heptane does not currently support pointers, it is sufficient to analyze register contents to compute the addresses of load and store instructions and thus no analysis of memory contents is required. The possible abstract values for a register are: $\perp$, $\top$ or an interval of addresses. Value $\perp$ represents an invalid register content; it is used as an initial register value for the first instruction of every basic block. Value $\top$ represents a correct but unknown value (any possible value but $\perp$). We use $\top$ to specify the contents of a register after a load from memory because memory contents are not analyzed. Data flow equations, detailed in [9] for an early version of the data address analysis, are defined for all instructions, to specify the impact of the instruction on the registers abstract values. When the address of a static variable is loaded into a register (e.g. when loading the start address of an array in a register), the symbol table is used to determine the corresponding address interval; for local data the entire stack frame is used as interval.

Cache analysis

The *cache analysis* is the most developed analysis in *HeptaneAnalysis*. It allows to analyze set-associative[3] instruction and data cache hierarchies under different cache replacement policies (LRU, PLRU, FIFO, MRU, Random).

[3] Direct mapped caches and fully-associative caches are specific cases of set-associative caches.

The cache analysis of a cache level in *HeptaneAnalysis* is an implementation of the *Must*, *May* and *Persistence* analyses [27, 13] based on abstract interpretation [7]. The principle of the analyses is to statically associate a Cache Hit/Miss Classification (CHMC) for every memory reference. The CHMC defines for each reference its worst-case behavior with respect to the cache under analysis (e.g. the CHMC is set to *always-hit* only when it is guaranteed the reference will always result in a cache hit, regardless of the execution path followed at run-time).

The *Must*, *May* and *Persistence* analyses calculate for every basic block and every call context a corresponding *Abstract Cache State (ACS)* whose semantics depend on the analysis. For example, the ACS for the *Must* analysis at a given program point contains the addresses of the memory blocks that are guaranteed to be in the cache at that point. The structure of ACS depends on the cache replacement policy. For the most predictable replacement policy LRU [23], the associativity of the ACS is the same as the one of the concrete cache. For LRU replacement, the position of a memory block in a set and a dataflow equations depends on the type of analysis (*Must*, *May* and *Persistence*); for example, for the *Must* analysis, a memory block in the *Must* ACS has an age that is higher than or equal to the age of the block in the LRU stack.

To analyze cache replacement policies other than LRU, we have implemented a method using the metrics proposed in [23] that characterize the life time of references in a cache for different replacement policies. These metrics are used to set the associativity of the ACS to a value lower than the associativity of the concrete cache state for the *Must* and *Persistence* analyses. For example, for a cache with a random replacement policy, any memory block in a set may be replaced upon cache replacement, making such caches equivalent to a direct-mapped cache for the *Must* and *Persistence* analyses regarding CHMC classification.

To analyze cache hierarchies, we have implemented the method proposed in [10, 14] that introduces in the cache analysis a *Cache Access Classification* (CAC) to take into account the filtering effect of the previous cache level in the hierarchy.

For data caches, we assume a *write-through no-write-allocate* policy. A *write-through* policy was analyzed because it is easier to analyze than a *write-back* strategy (in contrast to *write-back* caches, it is easy to know for *write-through* caches when memory accesses will take place). A *no-write-allocate* policy is assumed because *write-through no-write-allocate* is the most common configuration found for *write-through* caches.

Pipeline analysis

The pipeline analysis for all supported architectures, currently considers a simple in-order pipeline free from timing anomalies, [18] with one cycle per stage except for the fetch and memory stages where the results of the cache analysis are taken into account.

Interactions between analyses

The initial CFG is gradually enriched with the results of the analyses, thanks to the attribute attachment facilities provided by the *cfglib* library. For instance, the instruction cache analysis attaches to every instruction and cache level an attribute defining the instruction CHMC and CAC. The attribute will be used by the pipeline analysis to compute the worst-case execution time of every basic block, that will be attached as an attribute to every basic block. The WCET calculation phase will use WCETs of basic blocks to compute the overall WCET. Every analysis has a dedicated method that checks the presence of its inputs (i.e. the corresponding attributes were attached by the analyses executed before). Checking is based on attribute names. For example, the WCET calculation analysis checks that all loops have an attribute named *maxiter* attached.

4 Using Heptane

Heptane can be installed on Linux and Mac OS X. For its installation, Heptane has some requirements: *libxml2* and an ILP solver (*lpsolve* or IBM CPLEX). To use the optional analyses of Heptane described at the end of this section (graphical output, documentation extraction), *dot*, *epstopdf* and *Doxygen* have to be installed as well. The rest of this section gives an overview of how to use *HeptaneAnalysis*, after CFG extraction (complete documentation can be found in the Heptane web page).

HeptaneAnalysis takes as parameter an XML configuration file describing the architecture under analysis and the analyses to be applied. The analyses operate on a CFG produced by the CFG extractor *HeptaneExtract*. The extracted CFG, whose format is not presented here for space considerations, is a human-readable file with one XML tag per object (program, function, basic block, edge, loop, instruction) with attributes attached when relevant (loop bounds, instruction addresses, etc). A unique identifier is assigned to each object as an XML attribute for deserialization.

4.1 Description of the architecture under analysis

In case the user wishes to use custom parameters describing the architecture or the analyses to perform, a template of configuration file is available in the *config_files* directory. The configuration file starts with an INPUTOUTPUTDIR XML tag to set the working directory:

```
<INPUTOUTPUTDIR name="HEPTANE_ROOT/benchmarks/simple"/>
```

This directory should contain the XML files resulting from CFG extraction and, will contain the result files of the analyses. Then the processor type, the cache hierarchy and the memory have to be described as follows:

```
<ARCHITECTURE>
  <TARGET NAME="MIPS" ENDIANNESS="BIG"/>
  <CACHE nbsets="32" nbways="2" cachelinesize="32"
  replacement_policy="LRU" type="icache" level="1" latency="1"/>
  <CACHE nbsets="64" nbways="8" cachelinesize="64"
  replacement_policy="LRU" type="icache" level="2" latency="10"/>
  <CACHE nbsets="32" nbways="2" cachelinesize="32"
  replacement_policy="LRU" type="dcache" level="1" latency="1"/>
  <CACHE nbsets="64" nbways="8" cachelinesize="64"
  replacement_policy="LRU" type="dcache" level="2" latency="10"/>
  <MEMORY load_latency="100" store_latency="100"/>
</ARCHITECTURE>
```

The user has to provide the target processor (i.e. MIPS or ARM) and the architecture endianness, the memory latency for the loads and stores in processor cycles and for each level of cache:

- the number of sets, the number of ways (associativity), the size of a cache line in Bytes;
- the replacement policy (LRU, PLRU, MRU, FIFO or RANDOM):
- the type of the cache (icache/picache for an instruction cache and dcache/pdcache for a data cache). Letter p indicates that the cache is a perfect cache that always hits. In this case, only one cache level can be specified and the other parameters except the hit latency are ignored.

- the level in the hierarchy (1 for L1, 2 for L2 and so on)
- the hit latency in processor cycles

4.2 Using the main analyses

The main part of the configuration file is the description of the analyses the user wants to perform. They have to be defined inside an ANALYSIS XML Tag and they are applied in their order of appearance in the configuration file. In case an analysis A relies on the results of previously applied analyses, analysis A checks the presence of all its mandatory input information. Each analysis has three common parameters:

- *keepresults* set to true to keep CFGs and the results of the analysis in memory and *false* otherwise
- *input_file* set to empty string in case the user wants to reuse the results of a previous analysis, or to a XML file previously exported by an analysis,
- *output_file* set to empty string in case the user does not want to serialize the modifications done by the analysis or to *file.xml* to store it in the mentioned file

The first analysis to apply is to fix the entry point of the program and to compute the calling context of functions accordingly. This is illustrated below:

```
<ENTRYPOINT keepresults="on" input_file="simple.xml" output_file=""
 entrypointname="main"/>
```

In case the user wants to perform a data cache analysis, the data address analysis has to be performed beforehand, to determine the memory addresses of load and store instructions. This analysis takes as parameter the address of the stack pointer before the execution of the entry point, as shown below:

```
<DATAADDRESS keepresults="on" input_file="" output_file="" sp="7FE000"/>
```

For the cache analysis, an analysis has to be defined for each cache[4] of the architecture, and the different levels of caches have to be analyzed in order (i.e. L1 before L2 and so on). The name of the analyses are ICACHE and DCACHE for an instruction cache and a data cache respectively. The analysis has to indicate the cache level and if the *must*, *persistence* and *may* analyses has to be performed, as shown below for a L1 instruction cache:

```
<ICACHE keepresults="true" input_file ="" output_file =""
 level="1" must="on" persistence="on" may="off"/>
```

The pipeline analysis can be called once the caches have been analyzed only. The analysis is defined by a PIPELINE XML tag, with only the common parameters. Finally, WCET estimation is performed by the IPET analysis:

```
<IPET keepresults="on" input_file ="" output_file =""
 solver="lp_solve" attach_WCET_info="true" generate_node_freq="true"/>
```

The declaration should provide the name of the ILP solver (i.e. *lp_solve* or *cplex*). Furthermore, the user should indicate if the estimated WCET of the program as well as the estimated frequency of each basic block have to be kept for the next analyses.

[4] In case of a perfect cache, the analysis has to be performed as well and the corresponding cache level has to be set to 1.

4.3 Other useful analyses

HeptaneAnalysis can provide useful feedback to the user through additional analyses. The first one is the cache statistics analysis, that provides for each cache the number of references, the number of hits and misses along the longest path identified by IPET. The analysis is defined by a CACHESTATISTICS XML tag with only the common parameters.

The user may want to have a graphical representation of the program. *HeptaneAnalysis* can generate a *pdf* file representing the CFG of the benchmark by inserting a DOTPRINT XML tag with only the common parameters.

The user can also print in text format the estimated WCET, the CFG, the call graph, and the loop structure and loop bounds of the benchmark. This is achieved by the SIMPLEPRINT analysis:

```
<SIMPLEPRINT keepresults="on" input_file="" output_file=""
 printWCETinfo="on" printcfg="off"
 printcallgraph="off" printloopnest="off"/>
```

Finally, *HeptaneAnalysis* offers the possibility to display the path identified by IPET as the longest path on the source code, with the estimated execution frequency of each line of code in a HTML file (several times in presence of different calling contexts). To do so, a mapping between the source and the binary[5] is first performed by calling the *addr2line* tool on the binary. In the configuration file, this is performed in two steps: (*i*) the mapping and (*ii*) the production of the HTML file:

```
<CODELINE keepresults="on" input_file="" output_file=""
 binaryfile="simple.exe"
 addr2lineCommand="HEPTANE_ROOT/CROSS_COMPILERS/MIPS/bin/mips-addr2line"
/>
<HTMLPRINT keepresults="on" input_file ="" output_file =""
 colorize="true" html_file="simple.html"/>
```

5 Performance of Heptane

Table 5 gives the analysis time in seconds (total and per analysis) and WCET in cycles for the set of Mälardalen benchmarks[6] that are supported by Heptane. The analysis was performed on MIPS code, with two levels of instruction and data caches with the same structure (2-way set-associative cache with 32 sets and 32-byte blocks for L1, 8-way set-associative cache with 64 sets and 64-byte blocks for L2). The analysis was executed on an Intel Core i7 quad-core . The solver used by the IPET analysis is lp_solve 5.5.2.

The analysis time of all benchmarks except the biggest ones (*nsichneu* and *statemate*) is very low (below one second and up to 3.1 seconds for *fft*). The worst-case analysis time was observed on *nsichneu* and was clearly dominated by the cache analysis time.

6 Extending Heptane

The software architecture of Heptane was designed to be modular and extensible. In addition, it was decided to keep in the main branch of Heptane only the analyses that we think can be useful to a wide audience. Other analyses, developed over the year to

[5] The benchmark has to be compiled with the *-ggdb gcc* option.
[6] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Table 1 Analysis time (seconds) per analysis steps and resulting WCET (cycles).

benchmark	icache (L1)	icache (L2)	dcache (L1)	dcache (L2)	IPET	Total	WCET
bs	0.005	0.009	0.007	0.010	0.005	0.079	4020
bsort100	0.010	0.016	0.009	0.014	0.006	0.150	5812114
crc	0.038	0.049	0.034	0.058	0.013	0.630	1782419
expint	0.012	0.024	0.012	0.020	0.008	0.224	647343
fft	0.261	0.297	0.136	0.175	0.063	3.144	1253683
fibcall	0.002	0.004	0.003	0.005	0.003	0.050	14023
insertsort	0.003	0.006	0.004	0.008	0.004	0.090	52355
jfdctint	0.017	0.019	0.022	0.023	0.011	1.026	100331
lcdnum	0.074	0.116	0.021	0.057	0.009	0.366	9106
ludcmp	0.068	0.085	0.324	0.283	0.016	1.289	381353
matmult	0.014	0.023	0.017	0.027	0.008	0.317	1795585
minver	0.039	0.069	0.044	0.060	0.020	2.784	66835
ns	0.007	0.013	0.011	0.015	0.005	0.153	146208
nsichneu	18.314	21.468	2.815	3.250	0.874	52.775	515015
qurt	0.070	0.089	0.059	0.080	0.027	1.599	111554
select	0.068	0.073	0.023	0.034	0.014	0.547	91798
sqrt	0.012	0.016	0.010	0.018	0.011	0.183	20159
statemate	3.563	5.643	0.322	0.391	0.141	10.730	229575
ud	0.042	0.068	0.096	0.101	0.015	0.674	210770

experiment research techniques are kept in dedicated folders (non exhaustively: analysis of shared cache interference [8, 21], analysis of cache hierarchy management policies [11], cache related preemption delay estimation [19, 22], cache-partitioning [15], traceability of flow information [16], static probabilistic WCET analysis [1, 12] analysis of code caches in just-in-time compilers [3]).

Each analysis is located in a distinct directory and inherits from an *Analysis* base class such that the developer of a new analysis respects good practice (checks the presence of the inputs required by the analyses and cleans up internal attributes). A dummy analysis *DummyAnalysis* is provided as an example. Moreover, the *cfglib* library, central for implementing analyses, is well documented.

7 Conclusion and future work

We have described in this paper Heptane, an open-source software program for WCET estimation, that estimates WCETs from program binaries. Heptane has reached over the years sufficient reliability to be used by researchers external to our group. The most advanced analyses integrated in Heptane are the static cache analyses (support for multiple replacement policies and cache hierarchies). Its weaknesses, that we hope to address in the future, are its lack for automatic extraction of loop bounds, its small number of target processors, and its pessimistic albeit safe data address analysis. Our current work on the tool is to address the latter issue, in particular for stack-allocated data and accesses to arrays. Another direction is to use the XML format used by our open-source CFG management library *cfglib* as an exchange format between tools, to gather the best analysis techniques from different groups into a common WCET estimation infrastructure.

Acknowledgments. Heptane is a collective work to which many people beyond the authors of this paper have participated. The authors would like to thank the following people who actively participated in the development and test of the current version of Heptane: Thomas Piquet, Benjamin Lesage, Erven Rohou, François Joulaud, Nicolas Kiss and Loïc Besnard, as well as developers of older versions, who helped setting up a satisfactory software architecture for the tool: Antoine Colin, Jean-François Deverge and Matthieu Avila.

References

1 Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 266–275, 2014.

2 aiT. `https://www.absint.com/ait/`.

3 Adnan Bouakaz, Isabelle Puaut, and Erven Rohou. Predictable binary code cache: A first step towards reconciling predictability and just-in-time compilation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 223–232, 2011.

4 Bound-T. `http://www.bound-t.com/`.

5 Chronos. `http://www.comp.nus.edu.sg/~rpembed/chronos/`.

6 Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2/3):249–274, May 2000.

7 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

8 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 68–77, 2009.

9 Damien Hardy and Isabelle Puaut. Predictable code and data paging for real time systems. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 266–275, 2008.

10 Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, December 2008*, pages 456–466, 2008.

11 Damien Hardy and Isabelle Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture*, 57(7):677–694, 2011. Special Issue on Worst-Case Execution-Time Analysis. `doi:10.1016/j.sysarc.2010.08.007`.

12 Damien Hardy and Isabelle Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51(2):128–152, 2015.

13 B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, April 2011. `doi:10.1109/RTAS.2011.27`.

14 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative data caches. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2009. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2009.2283`.

15 Benjamin Lesage, Isabelle Puaut, and André Seznec. PRETI: partitioned real-time shared cache for mixed-criticality real-time systems. In *20th International Conference on Real-Time and Network Systems, RTNS'12, Pont a Mousson, France – November 8-9, 2012*, pages 171–180, 2012.

16 Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 217–226, 2015.

17 Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In Richard Gerber and Thomas Marlowe, editors, *LCTES'95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, volume 30, pages 88–98, New York, NY, USA, 1995. `doi:10.1145/216636.216666`.

18 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.

19 José Marinho, Vincent Nélis, Stefan M. Petters, and Isabelle Puaut. Preemption delay analysis for floating non-preemptive region scheduling. In *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 497–502, 2012.

20 OTAWA. `http://www.otawa.fr/`.

21 Dumitru Potop-Butucaru and Isabelle Puaut. Integrated worst-case execution time estimation of multicore applications. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 21–31. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013. `doi:10.4230/OASIcs.WCET.2013.21`.

22 Syed Aftab Rashid, Geoffrey Nelissen, Damien Hardy, Benny Akesson, Isabelle Puaut, and Eduardo Tovar. Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 262–272, 2016.

23 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. `doi:10.1007/s11241-007-9032-3`.

24 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. URL: `http://www.jopdesign.com/doc/t-crest-jnl.pdf`, `doi:10.1016/j.sysarc.2015.04.002`.

25 Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, November 1996.

26 SWEET. `http://www.mrtc.mdh.se/projects/wcet/sweet/`.

27 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000.

28 Tutor 2016. `https://tutor2016.inria.fr/`.

29 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. `doi:10.1145/1347375.1347389`.