

parMERASA

Multi-Core Execution of Parallelised Hard Real-time Applications Supporting Analysability

D3.12 - Report on support of tools for case studies

Nature:	Report
Dissemination Level:	Public
Due date of deliverable:	September 30, 2014
Actual submission:	September 30, 2014
Responsible beneficiary:	RPT
Responsible person:	Nicholas Lay, Dave George

Grant Agreement number:	FP7-287519
Project acronym:	parMERASA
Project title:	Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability
Project website address:	http://www.parmerasa.eu
Funding Scheme:	STREP: SEVENTH FRAMEWORK PROGRAMME THEME ICT – 2011.3.4 Computing Systems
Date of latest version of Annex I against which the assessment will be made:	July 14, 2011
Project start:	October 1, 2011
Duration:	36 month
Periodic report:	Third Periodic Report
Period covered:	2013-10-01 to 2013-09-30
Project coordinator name, title and organization:	Prof. Dr. Theo Ungerer, University of Augsburg Tel: + 49-821-598-2350 Fax: + 49-821-598-2359 Email: ungerer@informatik.uni-augsburg.de

Release Approval:

Name	Role	Date
P. Sainrat	WP3 leader	30/09/2014
T. Ungerer	Project co-ordinator	30/09/2014

DELIVERABLE SUMMARY

Deliverable 3.12, “Report on support of tools for case studies” reports the work done in Work Package 3 (WP3) to reach milestone MS7 (see DoW, sect. 1.3.3, p. 45).

Tasks performed

The objectives of WP3 in period 3 concern the tool support for WCET analysis, verification, and profiling of parallel programs.

- Task 3.4 “Techniques and WCET analysis tools supporting the parMERASA architecture and parallel applications” concerns the investigation of techniques for WCET analysis of parallel SW design patterns and its implementation of an automatic static WCET analysis process for parallel application in the OTAWA toolset.
- Task 3.5 “On target verification tools for parallel programs” and Task 3.6 “On target profiling of parallel programs” concern tools to be developed by RPT to assist during the parallelisation process and to verify the results of parallelised programs.

Tasks 3.4, 3.5 and 3.6 were all executed from month 10 to month 30. Prototypes of the analysis tools were previously delivered in Deliverables D3.8-D3.11.

- Task 3.7 “Analysis of pilot studies and Open Source OTAWA tool” (month 31 – month 36) concerns the WCET analysis of pilot studies, use of on target verification and profiling tools on case studies and public release of the OTAWA tool under an Open Source license.

Measurable objectives:

The measurable objectives of WP3 were:

- O3: Develop on target verification tools, in particular WCET analysis tools with less than 25% pessimism on WCET estimates of parallel programs. Further tools to be developed concern code coverage and memory analysis as well as parallel program profiling and visualisation. This objective will be demonstrated by review of evidence that can support a certification argument and by applying the tools to the case studies.

The Objective “O3: Develop on target verification tools, in particular WCET analysis tools with less than 25% pessimism on WCET estimates of parallel programs.” has been reached, although the “25% pessimism on WCET estimates of parallel programs” could not be demonstrated. A discussion of possible sources of pessimism is given in Section 2.2.5 of this document.

Tools developed by RPT concern code coverage and memory analysis as well as parallel program profiling and visualisation. This objective was demonstrated by review of evidence that can support a certification argument and by applying the tools to the case studies.

- O7: Contribute to Open Source software. The software developed at the universities, i.e. the static WCET tool OTAWA of WP3, will be made publicly available under an Open Source license at the end of the parMERASA project.

The Objective “O7: Contribute to Open Source software.” is fulfilled by the static WCET tool OTAWA, which is made publicly available under an Open Source license at the end of the parMERASA project.

Milestones

Milestone 7 “Optimisation, Refinement and Evaluation” is the outcome of Task 3.7, documented in Deliverable D3.12. The milestone concerns:

- WCET tools perform automatic WCET analysis of parallel programs
- OTAWA tool publicly available under an Open Source license
- Assessment if the WCET analysis tools reach the target of less than 25% pessimism on WCET estimates of parallel programs
- Verification and profiling tools support program development

All objectives and the milestone have been accomplished.

TABLE OF CONTENTS

1	Introduction.....	8
2	Tool development during the project	9
2.1	Hybrid measurement-based WCET tool (RapiTime)	9
2.1.1	WCET calculation for a parallel application.....	10
2.1.2	Automatic identification of synchronisation primitives.....	11
2.1.3	Measurement of waiting and execution times for synchronisation primitives	12
2.1.4	Calculation of worst-case waiting and execution times for synchronisation primitives.....	13
2.1.5	Integration with the parMERASA simulator and build environment	16
2.2	Static WCET analysis tool (OTAWA)	17
2.2.1	Improved flow analysis.....	17
2.2.2	Data cache analysis.....	18
2.2.3	Modelling of the parMERASA architecture	18
2.2.4	Analysis of parallel programs	18
2.2.5	Analysis of the possible pessimism of WCET estimates	19
2.3	Verification tools	21
2.3.1	On-target structural code coverage tool.....	22
2.3.2	Tool to verify equivalence in both sequential and parallel versions of the software ...	23
2.4	Visualization and profiling tools	27
2.4.1	Visualisation tool to assist with parallelisation of existing sequential software.....	28
2.4.2	Parallel system visualisation and profiling tool	29
3	WCET analysis of pilot studies.....	33
3.1	3D path planning (Honeywell).....	33
3.1.1	Structure of the application and approach to its static WCET analysis (OTAWA)	33
3.1.2	Provisional OTAWA analysis and optimisation of the application	34
3.1.3	Final OTAWA results.....	36
3.1.4	Analysis with RapiTime.....	36
3.1.5	Final RapiTime results.....	37
3.1.6	Comparison of OTAWA and RapiTime results.....	37
3.1.7	Evaluation and lessons learned	38
3.2	Stereo navigation (Honeywell)	39
3.2.1	Analysis with OTAWA: state of progress	39
3.2.2	Analysis with RapiTime.....	39

3.2.3	Final RapiTime results.....	40
3.2.4	Comparison of OTAWA and RapiTime results.....	41
3.2.5	Evaluation and lessons learned	41
3.3	Diesel engine controller (DENSO).....	41
3.3.1	Analysis with OTAWA	42
3.3.2	Analysis with RapiTime	42
3.3.3	Final RapiTime results.....	42
3.3.4	Comparison of OTAWA and RapiTime results.....	42
3.3.5	Evaluation and lessons learned	42
3.4	Dynamic compaction (Bauer)	42
3.4.1	Structure of the application and approach to its static WCET analysis.....	43
3.4.2	Analysis with OTAWA	44
3.4.3	Analysis with RapiTime	44
3.4.4	Final RapiTime results.....	45
3.4.5	Comparison of OTAWA and RapiTime result	46
3.4.6	Evaluation and lessons learned	46
4	Conclusions on the support of tools for case studies	48

1 INTRODUCTION

This deliverable presents the final results of Work Package 3 (WP3), the primary goal of which was to provide tools for analysis and verification of the parallel applications (produced in Work Package 2). This document outlines the results of this work, including information on the extensions made to the applications (along with any new applications developed), and reports on the final results collected from the usage of those applications in the project. The work package consists of two partners; Rapita Systems (RPT) and the IRIT Research lab from the University of Toulouse (UPS).

While some degree of raw result data (including values from worst case execution time analysis of parallelised applications) is presented in this document, it serves to provide information and comparisons, rather than acting to show improvements made by the partners in work package 2. For the full results of the parallelisation of industrial applications, please see Deliverable 2.6 [1].

Development of tools to complete WCET and verification analysis has been driven primarily by requirements outlined by industrial partners in parMERASA. Full details of the initial requirements considered for our development of verification tools can be found in Deliverable 3.1 [2]. Our initial work on guidelines for creating parallel applications with a view to analysability can be found in Deliverable 3.2 [3].

Throughout the project, we have delivered a number of prototype versions of tools to all other project partners, allowing them to integrate verification and timing analysis into their software development methodologies used in the parMERASA project.

This document consists of the following sections:

- Tool development during the project
 - This section outlines the overall goals and work done towards creating WCET analysis and verification tools in the parMERASA project. This includes detailed information about the methodologies used to calculate parallel timing estimates, along with information on the other verification tools developed as part of the project.
- WCET analysis of pilot studies
 - Section 3 gives details on the results collected from analysis of the parallelised applications created in Work Package 2. Results are presented, but in less detail than those shown in Deliverable 2.6 [1], and with a focus on the comparison of the results obtained from the OTAWA and RapiTime tools. The aim of this document is to use a subset of the collected results as a point of discussion of the outcome of work on verification tools.
- Conclusions on the support of tools for case studies
 - The final section presents a conclusion on the success of the work undertaken in Work Package 3 (the creation of verification tools for parallel programs).

2 TOOL DEVELOPMENT DURING THE PROJECT

At the start of the parMERASA project, requirements for tools were identified to support the WCET analysis of the parallelised applications, and also to support the process of parallelisation and to verify the functional equivalence of the parallelised versions compared with the original sequential applications.

The requirements for these tools were originally outlined in Deliverable 3.1 [2], in which development of the following tools was proposed:

- Tool 1: Hybrid measurement-based WCET analysis tool
- Tool 2: Static WCET analysis tool
- Tool 3: On-target structural code coverage tool
- Tool 4: Memory and stack analysis tool
- Tool 5: Visualisation tool to assist with parallelisation of existing sequential software
- Tool 6: Parallel system visualisation and profiling tool

During the course of the project, it became apparent that there was a need for an additional tool to verify that the parallelised application exhibits the same properties as the original sequential version. The need for this tool was determined to be greater than the need for the memory and stack analysis tool, and consequently the decision was taken to develop the equivalence-checking tool instead. Consequently, the set of tools that have been developed to support the parMERASA project is as follows:

- Tool 1: Hybrid measurement-based WCET analysis tool (RapiTime)
- Tool 2: Static WCET analysis tool (OTAWA)
- Tool 3: On-target structural code coverage tool
- Tool 4: Tool to verify equivalence in both sequential and parallel versions of the software
- Tool 5: Visualisation tool to assist with parallelisation of existing sequential software
- Tool 6: Parallel system visualisation and profiling tool

Versions of these tools have been previously supplied as prototype deliverables, according to the project schedule. The following sections of this document summarize the development of these tools and explain their features and benefits within the scope of the parMERASA project.

2.1 Hybrid measurement-based WCET tool (RapiTime)

The hybrid measurement-based WCET tool for the parMERASA architecture has been developed by Rapita Systems, extending the commercially-available RapiTime tool [4].

RapiTime combines static analysis of an application's source code with measurements taken from execution of the code in its target environment to accurately determine the WCET for the application. This approach allows an accurate WCET value to be determined without the need to model the characteristics of the target environment. Thanks to this, it is not necessary to construct an accurate model of a new architecture (such as the simulated multi-core architecture used in the parMERASA project) in order for it to be analysable using RapiTime (or indeed, any of Rapita's tools).

The major addition to RapiTime for the parMERASA project was the introduction of timing analysis for the multi-core synchronisation primitives used in the parallelised applications developed by the industrial partners. These modifications are described in the following sections.

Additionally, RapiTime also supplies detailed information about the observed execution time of the application, based on the measurements collected as the code is executed. This allows the user to compare the differences between the maximum observed execution time and the calculated worst case, which can be useful in determining where pessimism exists and support the identification of refinements that can be used to reduce it.

The parMERASA project has also required a number of extensions to RVS (the Rapita Verification Suite) to support analysis of the simulator based system used in the project, with a specific focus on adapting RVS to work with the build system used. Once this integration process has been completed, the analysis of new applications using the platform or altering the configuration parameters of an application does not require large amounts of effort. This approach isn't unusual, as the measurement based approach used by RVS always requires integration with the build system and target architecture used to execute the code (in order to collect execution traces).

Versions of the tool have been supplied in these prototype deliverables:

- D3.3: Preliminary version of WCET Tools including Partial Support for Analysis of Parallel Programs
- D3.6: Intermediate version of WCET Tools Supporting Automatic Analysis of Parallel Programs
- D3.9: Final version of RapiTime WCET Tools Supporting Automatic Analysis of Parallel Programs

2.1.1 WCET calculation for a parallel application

As part of the parMERASA project, worst case timings were calculated for the applications parallelised as part of Work Package 2 using RapiTime, the timing tool provided with the Rapita Verification Suite (RVS). RapiTime takes a fundamentally different approach to calculating WCET compared to OTAWA by using a combination of static information about the code and traces collected from on-target execution of the program.

This difference in approach also causes a fundamental difference in the workflow of creating an analysis. Where an OTAWA analysis utilises the final software binary, an analysis with RapiTime hooks directly into the build system.

Figure 1 outlines the integration of RVS with a user's build system, including the instrumentation of the source code, linking of an instrumentation library and generation of traces from the instrumented binary.

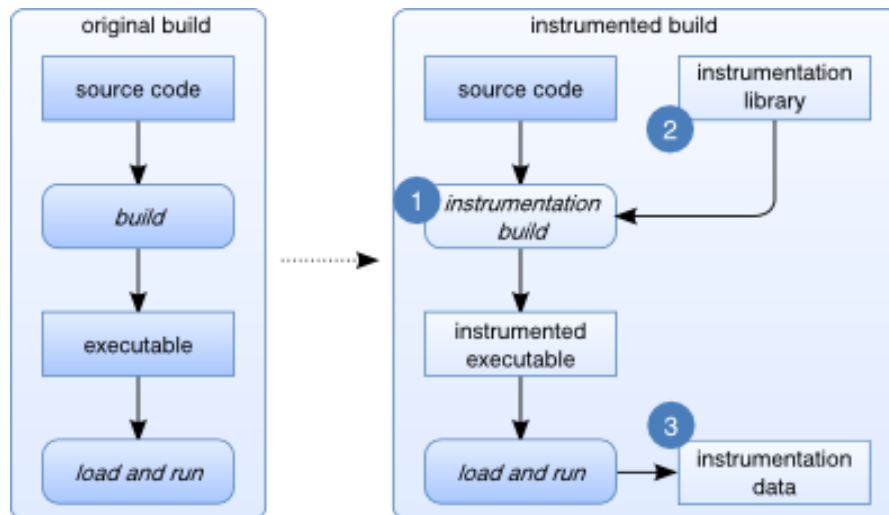


Figure 1: Integration of RVS

RapiTime works using the concept of **instrumentation points**, a series of calls which (when RapiTime is hooked into the build system) are automatically inserted into the code to be analysed at certain places. An instrumentation point calls a **trace library**, which is customised for the target the code is running on (be it a production system, a development board or a simulator). This library contains the relevant code needed to output trace data when called at an instrumentation point. Multiple methods for storing trace data exist, and the choice taken depends on the target platform being used. For more information on exactly how integration with the parMERASA applications was achieved, see section 2.1.5.

Once trace data is collected, the results are fed into the analysis tool, which, when combined with static information collected about the code, calculates the worst case timing of all functions which were covered while collecting traces. RVS also contains a number of other verification related functions which are covered in further detail in sections 2.3 and 2.4.

Calculating the WCET of a parallel application compared to a non-parallel application using RapiTime can differ slightly. When a parallel application is to be analysed, the final worst case timings become dependent on the possible worst case timings found in synchronisation primitives. This then means that multiple threads can contribute to the possible worst case execution time of certain paths through the program. Without using the parMERASA extensions, timing for the application simply relies on the **observed** time taken through synchronisation primitives, rather than doing detailed analysis of the possible calculated worst case time that any given thread may wait at a lock or barrier. This leads to an inaccurate result which may be either pessimistic or optimistic.

The insertion of instrumentation points into the source code necessarily results in a small increase in the execution time of the software. However, this increase can be measured (or calculated). In the case of the integration with the parMERASA simulator, the execution of a single instrumentation point is known to take exactly one clock cycle. The overhead can then be subtracted from the measured execution times during the analysis of the timing trace data.

2.1.2 Automatic identification of synchronisation primitives

RVS does not require the user to provide any annotations to user code in order to perform an analysis. However, annotations can be added later to refine the analysis. Based on this, RVS is required to automatically detect and understand the constructs contained within a user's code,

which allows RVS to calculate a worst case execution time based on knowledge of the structure of the code and the measurements obtained from a run-time trace.

When extending RVS to support WCET analysis of multi-core synchronisation primitives, we needed to extend our parser to collection information about the locks and barriers present in the code.

We have now extended our parser to support the identification of particular functions as barriers, locks or unlocks based on parameters provided by the user. Any function can be designated to be one of these special functions, however, the function must have a reference to a global variable in its parameter list. This allows the parser to link invocations of the lock or barrier between threads.

2.1.3 Measurement of waiting and execution times for synchronisation primitives

RVS' instrumenters (for C, C++ and Ada) have been modified to add 'instrumentation points' around the calls to synchronisation primitives. These instrumentation points call a function in a system specific 'trace library', which outputs trace data. This data then allows measurement of the time a task spends waiting at a synchronisation primitive. Each instrumentation point encodes an identifier, which indicates the position in the source code, and a timestamp.

For a critical section, instrumentation is inserted before and after the call to the lock function and also before the call to the unlock function. For a barrier, instrumentation is inserted before and after the call to the barrier function.

At the analysis stage, the time taken to go through every lock and every barrier transition is collected and analysed. This allows RVS to perform an analysis of the worst case waiting and execution time through any given lock or barrier transition. This is a significant development from the project. Careful analysis of the barrier waiting times ensures that the WCET results are not overly pessimistic. This is described in the next section.

The information gathered is included in the resulting RVS report file. The RVS Report Viewer has also been modified for parMERASA, providing new sections to display waiting and execution times for critical sections and barriers.

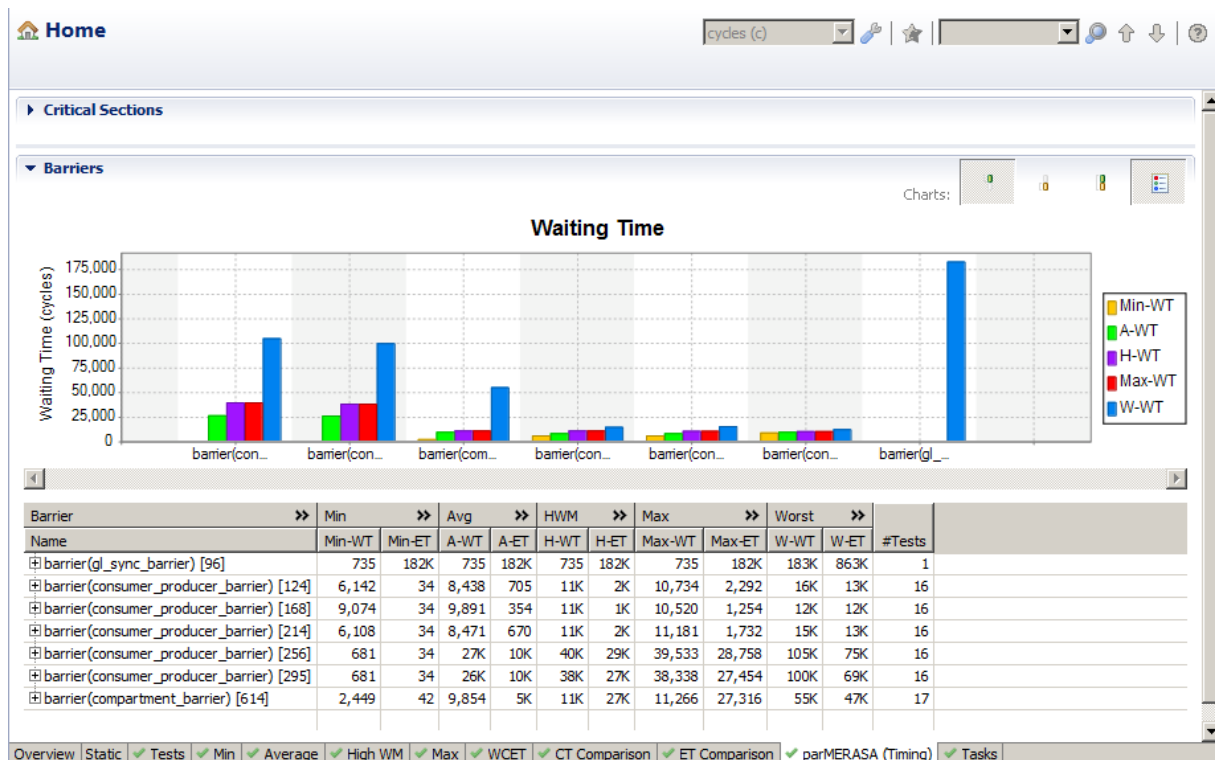


Figure 2: The parMERASA (Timing) display in the RVS Report Viewer

Figure 2 displays the parMERASA timing tab in the RVS Report Viewer. This particular example contains data from the 16 core configuration of Honeywell’s 3D Path Planning application, displaying minimum, average, high watermark, max and worst case timings and waiting times for all barriers found in the code.

2.1.4 Calculation of worst-case waiting and execution times for synchronisation primitives

Using the measurements collected from execution of the application, the worst-case waiting time and execution times for each synchronisation primitive can be calculated. The calculation of these values is based on the methods outlined in Deliverable 3.1: “Requirements on WCET Analysis and Verification Tools”.

However, some methods of timing analysis have changed slightly over the course of the project, specifically:

- Mutexes and Locks are analysed in fundamentally the same way as outlined in D3.1, with the timing between the lock and unlock of the given mutex providing the information required to generate a worst case estimate of timing and waiting time for the given lock.
- Barrier analysis evolved through two different methodologies during our development of the RVS parMERASA extensions. Our initial implementation of barrier WCET analysis was much more generalised but also significantly more pessimistic. While this did have its uses, it was ultimately a far too pessimistic number and didn’t fit perfectly with the requirements of the application providers. The improved technique is described below.
- Analysis of event dispatched based timing was dropped as it was not being used by the application partners when creating their parallelised implementations of applications.

The fundamental methodology used by RapiTime to analyse worst case times for barriers has changed throughout the course of the project.

Initially, our intention for worst case values for barriers was to provide a generalised but pessimistic approach, giving a single WCET value for each barrier, based on the calculated worst possible time among **all possible transitions from another barrier to the barrier being analysed**. So, the possible routes from all available barriers in the program to the barrier in question were analysed, and the calculated worst case time was used as the WCET value for any given barrier.

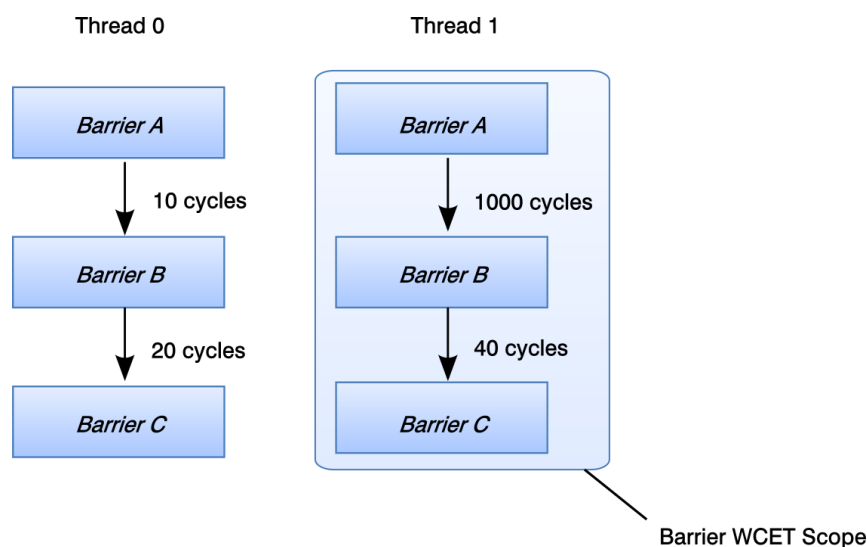


Figure 3: Initial barrier timing analysis methodology

Figure 3 outlines the basis behind our initial analysis approach. We decided to display a *single value* for each barrier in the program, with a barrier defined as a single global variable structure that could be synchronised on multiple times. Based on this, for any given barrier, we considered the *worst possible path that any other thread could take to reach a synchronisation point with this barrier throughout the entire program*. In the case of the example in Figure 3, the worst case time of Barrier B would be 1000 cycles, even if we only cared about the transition between Barrier B and C, which is highly likely to have a significantly lower worst case time. It became clear that this value didn't effectively fit with the timing model presented from the static OTAWA analysis tools, so our fundamental methodology for calculating and displaying WCET for a barrier was changed.

In the final version of the parMERASA extensions, WCET for barriers is calculated based on **transitions**. It became clear during the course of application development that, while the initial approach was amenable to code using a very complicated set of barriers and locks interacting in complicated ways, the timings were far too pessimistic. Based on this, we changed the calculation methodology to be focused primarily on the worst case timing possible between **specific sets of barrier synchronisation points**, rather than all barrier calls which share the same global variable. This allows a far less pessimistic timing value, as we now have a restricted (and more accurate) representation of what other threads are doing when calculating the longest possible time which could be spend waiting at any given synchronisation point.

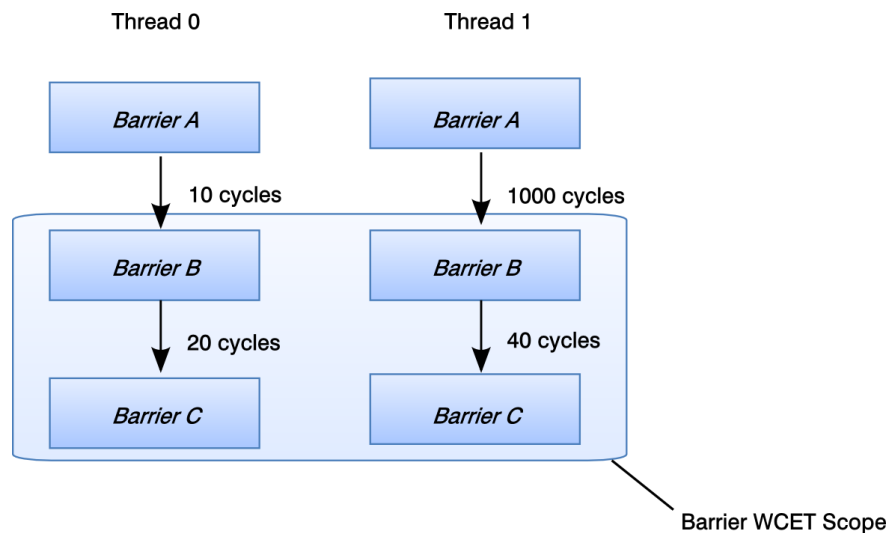


Figure 4: Updated barrier timing analysis methodology

Figure 4 shows the updated method. In this instance, we consider (and display the results for) each barrier **transition** in the code, constraining our analysis to particular parts of the program which may be of interest and also allowing us to be significantly less pessimistic.

However, this method does have one assumption which must hold to allow the analysis to complete. When a function is identified as a barrier waiting point in the analysis, the sequence of synchronisations using that barrier across all threads using it **must be the same**.

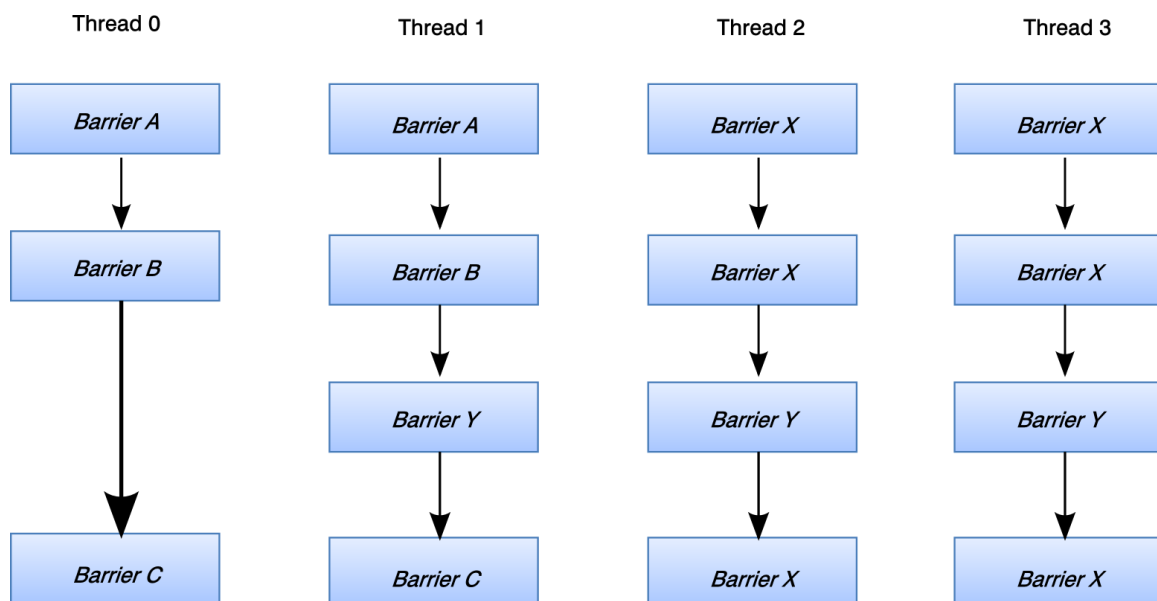


Figure 5: Unsupported barrier sequence

Figure 5 shows an unsupported sequence of barriers in a parallel application. If we were analysing threads 2 and 3 alone, analysis would complete successfully, as the two threads share the same barrier sequence, allowing us to easily know the state of execution on any other thread when we hit a synchronisation point. However, if we expand the scope of analysis to all threads, the calculation needed to ascertain where every other thread is when reaching any given synchronisation point becomes problematic. However, multiple barriers *can* be interleaved across threads, provided that the *overall sequence is the same across all threads*. Analysis of threads which do not share the same

barrier sequence has the potential to be implemented in future, though, allowing complex code using barriers as synchronisation points to be analysed while keeping pessimism in timing calculations low.

2.1.5 Integration with the parMERASA simulator and build environment

One of the most crucial parts of using RVS is the **integration stage**. Since RVS instruments an application by inserting calls into the application's source code, the RVS tools need to be hooked into the application's build system. This happens at three stages of the process, namely:

- Preprocessing - RVS instruments source code which has been pre-processed, eliminating the need for resolution of includes and imports at the instrumentation stage.
- Compilation - At the compilation stage, the files which would initially be compiled need to be switched out with the instrumented files, creating an alternative version of the binary with calls to an instrumentation library inserted.
- Linking - At the link stage, the target specific library for outputting trace data needs to be linked into the final binary. Also, it's at this stage that RVS generates structural information about the source code based on information collected at instrumentation time.

Naturally, all of these stages also needed to be performed on the applications being analysed as part of the parMERASA project.

As part of the parMERASA project, all applications developed by the industrial partners ran on the parMERASA simulator, a SoCLib/SystemC based simulator exposing a simulated platform consisting of multiple PowerPC based processor cores connected by a network-on-chip. Like a normal integration, we needed to consider the three stages outlined above.

The compilation stage for the parMERASA simulator itself was achieved by using the build system provided by SoCLib simulation environment. The compilation of software to be run on the simulator (i.e. The parallelised applications written by Honeywell, DENSO and Bauer) was built using a *make* based build system, with code being compiled using a *gcc* based PowerPC cross compiler. In this instance, RVS was hooked into the build system using the **RVS GCC Compiler Wrapper**. This tool replaces the compiler in use and is called directly by the build system. The wrapper then preprocesses and instruments the code, then passes it on to the real compiler for final compilation into an object. This allows instrumentation of code to be seamlessly accomplished without needing direct alterations to the build system. The instrumenter also generates *xsc* files (representing the structure of the source file) at this stage. These files are used later in order to perform structural analysis of the application.

At the link stage, the **RVS Linker Wrapper** is used, which operates in the same manner as the compiler wrapper, except that rather than instrumenting, it generates an *RVS Report* at the link stage, bringing together the structural information found in the instrumented source files into a single file, while also linking the trace library into the final application.

The final stage of integration into the parMERASA simulator involved two steps. Firstly, the simulator itself needed to be updated to support a tracing mechanism. Secondly, the RVS tracing library for the simulator needed to be written, providing the code to be inserted when an instrumentation point is found in the output application code.

To support the output of traces, the simulator required two new features to be added. Firstly, we added code to call the **Rapita RPZ library**. This library provides functions which can be called to output trace information in the RVS compressed trace file format, which offers very efficient compressed storage of trace data. Then, the simulator was configured to write an entry to the trace based on certain addresses being seen in the execution trace based on an address to ipoint mapping file. The mapping file was generated during the instrumentation/compilation process by adding the following code to the RVS instrumentation library:

```
#define ASM(X) ({__asm volatile(#X);})  
#define RVS_I(X) ASM(.globl _lbl_rvs_##X \n _lbl_rvs_##X: nop)
```

This code causes a global assembly label with a unique identifier to be added wherever an ipoint is required, followed by a single “nop” instruction (this is necessary to cause consecutive instrumentation points to have different addresses). After the build process completes, the binary is parsed, and all global assembly labels corresponding to an ipoint are added into the ipoint mapping file. The simulator is then executed and will output an element to the trace file when one of the addresses in the mapping file is encountered.

2.2 Static WCET analysis tool (OTAWA)

The static WCET analyser of the parMERASA project has been built on top of the OTAWA toolset. OTAWA includes several tools that can control static WCET analysis, as well as a library that can be used to develop additional functionalities or tools. It has been developed in the IRIT lab at UPS for about 10 years [1]. The initial version of OTAWA (at the beginning of the project) was able to perform WCET analysis on sequential code only and to model generic (parameterized) single-core architectures with instruction caches.

WCET analysis in OTAWA is done following the usual four-step process:

- a) *flow analysis*, that examines the binary code and builds the control flow graph (CFG) of the program;
- b) *global low-level analysis*, that consists in determining the behaviour of history-based components, such as caches – this step is based on abstract interpretation techniques
- c) *local low-level analysis*, that computes the worst-case execution cost of each basic block processed through the execution pipeline – this step uses an original approach developed at UPS
- d) *WCET computation* with the IPET technique, which solves an integer linear program that maximises the program execution time under control flow constraints.

OTAWA is freely distributed with oRange, its companion tool that determines contextual loop bounds [2].

Five kinds of extensions to OTAWA have been developed to fulfil the objectives of the parMERASA project. These are outlined in the following sections.

2.2.1 Improved flow analysis

This extension was required to meet the requirements from pilot studies. It includes the identification of conditional statements in the source and binary codes, as well as the implementation of additional flow annotation labels to specify detailed flow control (e.g. to specify that a given branch is always taken in a particular scenario of execution). This is now included in the parMERASA version of OTAWA.

2.2.2 Data cache analysis

The initial version of OTAWA did not support data caches. However, it has been found, from the preliminary analysis of applications, that data caches would be needed in the parMERASA architecture. Therefore, we had to design and implement appropriate analyses.

Data caches differ from instruction caches for two reasons. First, the address of an access to the data cache is often dynamic: it is stored in a register the content of which is determined by the execution of previous instructions. Determining this address at analysis time is relatively easy for accesses to global data. For accesses to stack data, it might be complex to determine absolute addresses, but identifying relative addresses is generally enough to draw conclusions on the cache behaviour. The problem is more complex for accesses to arrays (see below) or accesses to dynamically allocated data (for the latter, the problem is still not solved to the best of our knowledge). The second issue with data caches is that a single load/store instruction may access several data, e.g. when it belongs to a loop body. For example, it may successively loads the elements of an array. To be able to predict the behaviour of the cache, the analysis must determine the sequence of addresses generated by such an instruction. Our approach to solve both these issues generates a formal representation of dynamic address patterns. This representation is built from an analysis of the semantics of the binary code of the program. For example, for a load instruction located in a loop that scans an array, it determines that the addresses successively accessed by the load are in the form of $\text{tab} + 4*i$. From such a representation, it can be derived which cache lines will be referred to by successive loads and this allows a precise analysis of the data cache behaviour. This will be detailed in a coming publication.

2.2.3 Modelling of the parMERASA architecture

To support the WCET analysis of programs running on the parMERASA architecture, we had to develop a specific model for the two original components that have been designed in WP5: the network on chip (NoC) and the ODC² cache.

Two NoC structures have been considered in the project (tree- and mesh-like) and both are supported by OTAWA. This means that the NoC parameters (e.g., structure or local latencies) can be specified as part of the analysis and the worst-case latency of each memory access is computed by OTAWA, as shown in Deliverable3.5.

The ODC² data cache aims at managing data coherency among cores (threads) in a way that keeps timing analysis feasible. It requires shared data to be accessed inside critical sections and to be flushed to the main memory when a thread leaves such a critical section. The abstract-interpretation-based analysis of data caches has been modified to reflect this behaviour, as described in D3.5. This is also described in a paper submitted to the Journal on System Architecture [4].

2.2.4 Analysis of parallel programs

Another objective of the project was to develop an approach to the analysis of shared-memory parallel programs.

The focus has been put on two kinds of synchronisations: locks, to protect critical sections (and accesses to shared variables), and barriers to synchronise the progress of different threads. We have shown how the worst-case stall times (WCST) at such synchronisation primitives, assuming time-predictable scheduling schemes as implemented in WP4, can be estimated. These stall times are derived from partial WCETs (WCETs of threads on a short sub-path). To identify the useful sub-paths,

it is necessary to retrieve detailed information on competing threads. The user must provide such information as annotations.

The WCET analysis is then performed in several steps: parsing of the annotations, identification of stall times that need to be computed, identification of sub-paths that must be analysed, WCET analysis of the sub-paths, combination of partial WCETs to compute stall times, integration of stall times to the costs of basic blocks (WCC). This is illustrated in Figure 6. The first and last steps shown on the diagram are the same as for the WCET analysis of sequential programs.

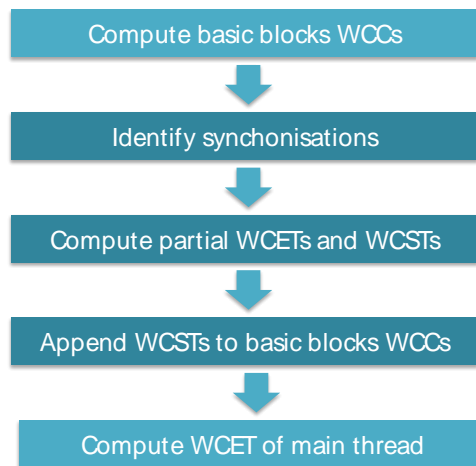


Figure 6: Analysis of parallel programs

2.2.5 Analysis of the possible pessimism of WCET estimates

Figure 7 shows how the pessimism of WCET estimates can be defined:

- Green vertical bars plot the real distribution of execution times, when all the possible execution paths are considered; the rightmost bar gives the real WCET (r-WCET).
- Measurement-based timing analysis exercises some possible paths, generally not all of them; to these paths correspond *observed execution times*.
- By construction, the static estimated WCET (e-WCET) is higher than r-WCET.
- It is tempting to consider the difference between the maximum observed execution time and e-WCET as pessimism. However, while no guarantee is given that the worst-case path has been exercised during measurements, *this difference does not mean anything about the accuracy of the WCET estimate*. At best, it could tell how far the WCET might be from what has been observed.
- Trivially, the real pessimism is the difference between r-WCET and e-WCET. Since the real WCET is unknown (otherwise, it would not have to be estimated), the real pessimism cannot be determined.

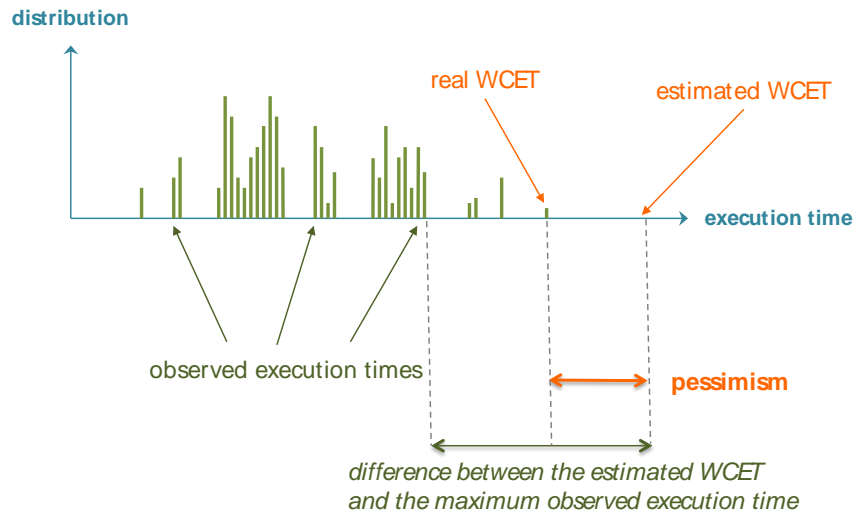


Figure 7: Observed execution time, estimated WCET and pessimism

Our approach to evaluate the possible pessimism of WCET estimates consists in computing a *lower bound* on the real WCET, denoted by l-WCET: it is the part of the estimated WCET that *can* happen for sure, according to the information known at analysis time. The pessimism is described by the difference between l-WCET and e-WCET.

We identified several possible sources of pessimism:

- **Incorrect/partial flow information:**
 - Unknown flow facts: it may happen that some flow facts are not known or specified at analysis time. For example, some input values may not be feasible in practice. The user should annotate it but this can be a fastidious and error-prone process. Ignoring that some input data have constrained values may lead to consider infeasible execution paths. However, we limit our scope to the pessimism generated by static analysis. Therefore, we assume that full information on the possible flow control is available at analysis time. As a result, our estimation of the pessimism of WCET estimates does not account for undefined flow facts.
 - Flow facts that cannot be expressed or accounted for in the WCET analysis: although detailed information on control flow restrictions may be known, it might be infeasible to express them as annotations due to limitations in the annotation format. For example, the flow fact annotation format used in OTAWA does not allow expressing that a branch is taken every third execution and not taken otherwise. Additionally, some flow facts that can be specified with annotations are eventually ignored when computing the WCET because they cannot be translated into integer linear constraints (IPET method). Again, we ignore such sources of pessimism and we assume that flow facts are all known, expressed and accounted for in the WCET estimation.
- **Over-approximation at analysis time:** because static WCET analysis does not unroll every possible execution path but instead derives properties that must hold for any path, it must over-approximate such properties (their value may be correct for some of the paths but might be an upper bound for the other paths). This over-approximation is a source of pessimism. The hardware mechanisms that require over-approximation at analysis time are

those that exhibit a history-dependent behaviour, such as pipelines, cache memories and branch predictors. In the parMERASA project, we consider simple cores that execute one instruction per cycle independently of the history. Thus they do not engender over-approximation. *Only cache memories* are likely to generate pessimism:

- Their behaviour depends on the execution path. As a result, cache analysis can predict the outcome of some of the accesses (and classify them as AlwaysHit or AlwaysMiss, or even Persistent) but cannot decide on some other accesses (labelled NotClassified). For undecided accesses, both possible outcomes (hit or miss) should be considered. However, thanks to the simple cores we consider, which are not subject to timing anomalies, the estimated WCET is safely computed by assuming a miss for such accesses.
- Accesses to the data cache often have a dynamic address that might be complex to determine at analysis time (as mentioned before, this is especially true for accesses to array elements). As a result, several categories of addresses are identified within the address analysis and some of them are marked as *Uncertain* or even *Undetermined*. An access with such an unknown address cannot be classified with respect to the cache behaviour (nor as a hit or a miss). In addition, it may break analysis results for other accesses since it cannot be determined which cache line will be replaced by this access. Imprecise address analysis can thus have disastrous effects on the cache behaviour analysis and, as a consequence, on the over-approximation of the WCET.

To quantify the over-approximation due to the analysis of cache memories, we compute the lower bound on the WCET (I-WCET) by considering that every undecided access is a hit. This is described in a coming publication that shows how analysis-related over-approximation can be estimated for various kinds of hardware schemes, including complex pipelines.

2.3 Verification tools

A major part of the parMERASA project has involved extending Rapita Systems' on-target verification tools which make up the rest of the Rapita Verification Suite, while also developing a new tool to aid in the parallelisation of legacy sequential software. The major verification tools provided as part of RVS are:

- **RapiCover:** a code coverage tool which provides information on statement, decision, loop and MC/DC¹ coverage.
- **RapiTask:** a tool that allows the user to view the captured traces from on target execution in a trace viewer, expanding function calls and showing detail on execution time and response times. When viewing the trace using RapiTask, users can click on events in the RVS Report Viewer and jump directly to where they occurred in the trace viewer.
- **RapiCheck:** a tool that provides *constraint checking*, a feature which allows users to define a set of properties about an application, then allows the user to check whether those properties held during execution, showing detailed information on any instances of failure.

¹ MC/DC refers to Modified Condition/Decision Coverage, a set of code coverage criteria typically used when testing software according to the DO-178B and DO-178C guidance (for the aerospace industry) or the ISO 26262 standard (automotive).

- **Rapita Dependency Tool:** developed entirely for parMERASA, this tool acts as a parallelisation aid for developers porting legacy applications to multi-core code. This tool parses the sequential code and displays global variable based dependencies between certain parts of the code based on user configuration.

2.3.1 On-target structural code coverage tool

This deliverable was provided by **RapiCover**, which has been updated to allow verification of code cover for multi-core applications. On top of extending the tool to support multi-core code, a number of features specific to parallelised applications have been added.

When a multi-threaded report containing coverage data is loaded into the RVS Report Viewer, users are now able to specifically see which parts of the code were executed and covered on each individual CPU core, as shown in Figure 8.

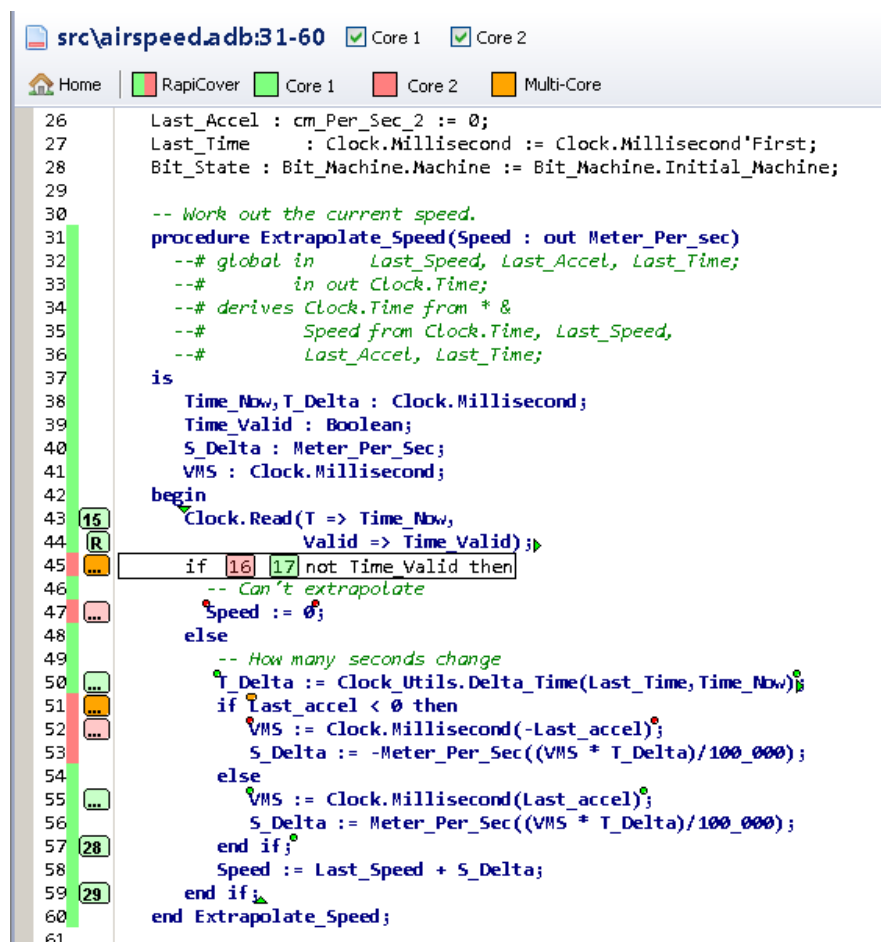


Figure 8: Multi-Core Coverage View

This allows a user to either see which parts of the code were covered across the entire application or which parts of the code were executed on a specific core. Like single core coverage, the multi-core extensions show comprehensive levels of cover including statement, decision, loop and MCDC coverage.

2.3.2 Tool to verify equivalence in both sequential and parallel versions of the software

Verification of equivalence between the sequential and parallel versions of the software developed for parMERASA is covered by **RapiCheck**, the constraint checking tool provided as part of the Rapita Verification Suite.

RapiCheck, like the other tools in RVS, has been extended to support parallel programs as part of the parMERASA project. The tool can now parse multiple traces from any number of cores, allowing the user to check that the parallelised version of their software is functionally equivalent to the sequential version. This is achieved by using *constraints*.

Constraints are the fundamental method by which RapiCheck checks equivalence between two applications. Constraints operate using a *timed finite automata* (TFA), which moves through steps based on symbols which map to events occurring in the program trace. When writing a constraint, a user can map an event of interest in the trace (such as a function entry or exit) to a symbol, which allows the automata to move through a transition based on the occurrence of that event. This functionality allows the creation of automata which can check that a series of events of importance occurred in the trace. Since the automata is timed, RapiCheck is also able to test and verify events of importance based on the time that they occurred.

Constraints can be used in two ways. RapiCheck contains a set of built-in, predefined constraints which can be used 'out of the box'. Using this library, the user simply writes a *.check* file containing calls to any pre-defined constraints as follows:

```
# functions f, g and h must appear in that order
constrain_order ("f","g","h");

# functions f, g, h, i and j must appear in that order
# and the sequence should not take more than 3ms
constrain_order_and_time ("f","g","h","i","j", "3ms");

# ET
execution_time ("f", "8us");

# RT
response_time ("f", "8us");
```

This example uses four constraints from the pre-defined constraint library. Here, we are instructing RapiCheck to check the following:

- That functions f, g and h execute in order.
- That functions f, g, h, i, and j execute in order and with a total end to end running time that is less than 5500 nanoseconds.
- That the execution time of function f does not exceed 8 microseconds.
- That the response time of function f does not exceed 8 microseconds.

RapiCheck will then parse the trace and collect data on how many times these constraints passed or failed during execution. Any failures will be reported in depth, including information on when they occurred (including the ability to jump directly to the failure point in the RapiTask Viewer) and how the automata failed.

When using constraints, automata for constraint checking can be directly defined using a domain specific language created for use with RapiCheck. This language allows the user to define automata directly, including their states, symbols and transitions. An example of a constraint defined using this language is shown below.

```
constraint jitter is timing
    "Test the jitter of a periodic task."
symbols
    A : start "periodic";
variables
    delta = 0;
    period = 4000;
    jitter = 1000;
    deltamin = period - jitter;
    deltamax = period + jitter;
begin
    state Start is
        when "A" => S0
    end state;

    state S0 is
        when "A" => {delta = WT - S0.WT}
            if ((delta < deltamin) || (delta > deltamax)) then
                NOK
            else
                OK
            end if;
    end state;
end constraint;
```

This constraint utilises the TFA model to check that jitter for a given task is within acceptable levels. Should a user wish to make a new TFA definition available to others, or to simply make the TFA simpler to use, they can update the language definition file to specify how the TFA should be built. This allows the use of the constraint in the same way as those defined in the built-in library.

As part of the parMERASA project, RapiCheck was first extended to support parallelised applications, and then tested with some example constraints on the parallelised applications provided by Work Package 2.

Our initial experiments with RapiCheck and a parallelised application used the 3D Path Planning application provided by Honeywell. While the project didn't allow for the use of RapiCheck as a fundamental part of the development process, we were able to create some examples of how the tool could be used in future to test the equivalence of a parallel application against a sequential one.

Our observations of the 3D Path Planning application shows us that it is data parallel, and uses multiple threads concurrently working on chunks of data which then synchronise at a common barrier. Based on this, our first experiment attempted to utilise RapiCheck to ensure all threads started with a certain time bound, with the bound being configurable.

▼ Constraints			
Overview of the constraints in the report			
Constraint >>			Pass / Fail >>
Name	Constraint Type	High Level Definition	Pass %
thread_start_bound	timing		100.000%
count_iterations	timing		-
check_iteration_counts	timing		100.000%

Figure 9: RapiCheck results with 3D Path Planning

Figure 9 displays the results when testing that all threads start within a particular bound. In this instance, the selected bound was large enough that all threads began executing within the defined bound. When changing the defined bound, RapiCheck successfully reports successes and failures based on the observed data. With a very high time bound, all threads easily start within the required time interval, causing all constraint tests to pass. If we lower the time, we see tests begin to fail, indicating that *some* of the threads are starting outside of the required bound. Setting the bound to be extremely small causes all tests to fail, with RapiCheck reporting information on the specific part of the automata which failed and reporting the timestamp.

Our second test attempted to ensure that the parallel computation of a grid of data executed as we would expect. Our observations tell us that the application calls the *gridlterate* function 6 times per core to compute data, then synchronises with the other threads at a barrier. Based on this, we wrote a constraint which checks that the expected amount of calls to *gridlterate* occurred before all threads synchronise. With a 16 core configuration, we would expect to see 96 calls to *gridlterate* before a synchronisation point. The constraint checks that the correct amount of calls occurred for the given core count. In Figure 9, setting the core count to 16 correctly verified that 96 executions of *gridlterate* took place before a synchronisation point. However, if we intentionally set the core count to be incorrect, the following result occurs:

▼ Constraints			
Overview of the constraints in the report			
Constraint >>			Pass / Fail >>
Name	Constraint Type	High Level Definition	Pass %
thread_start_bound	timing		100.000%
count_iterations	timing		-
check_iteration_counts	timing		0%

Figure 10: Failing constraints in RapiCheck

In Figure 10, RapiCheck correctly identifies that the number of calls to *gridlterate* before the barrier did not match what was expected, causing the test to fail. In this instance, we set the core count to 12, causing the automata to expect 72 calls to *gridlterate*, but 96 were observed.

If we drill down further into the failures, we can identify specific points in the execution where a given constraint failure occurred.

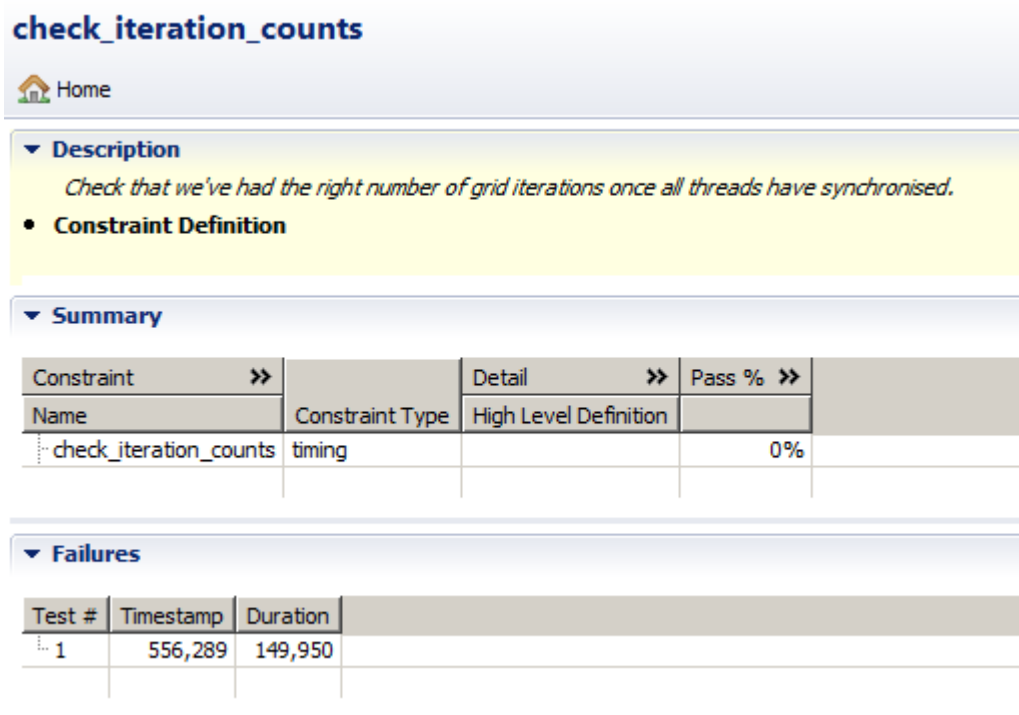


Figure 11: Summary of constraint failures

In Figure 11, we can see all instances of the constraint which failed, a specific timestamp of the failure, and the duration of the failure. Timing is initially displayed in clock cycles, but can be changed to multiple alternative units (such as nanoseconds, microseconds, milliseconds or seconds).

Further to this, we can also drill down into the specifics of any given constraint failure which occurred, as shown in Figure 12. Here we can see the symbols sent to the automata, the path taken through the automata that resulted in this particular failure, the symbols and variables in the program and the states and transitions in the automata with provided information on how many times they were tested.

Clicking on timestamps will also jump directly to the failure position in the RapiTask trace viewer, allowing easy debugging of constraint failures.

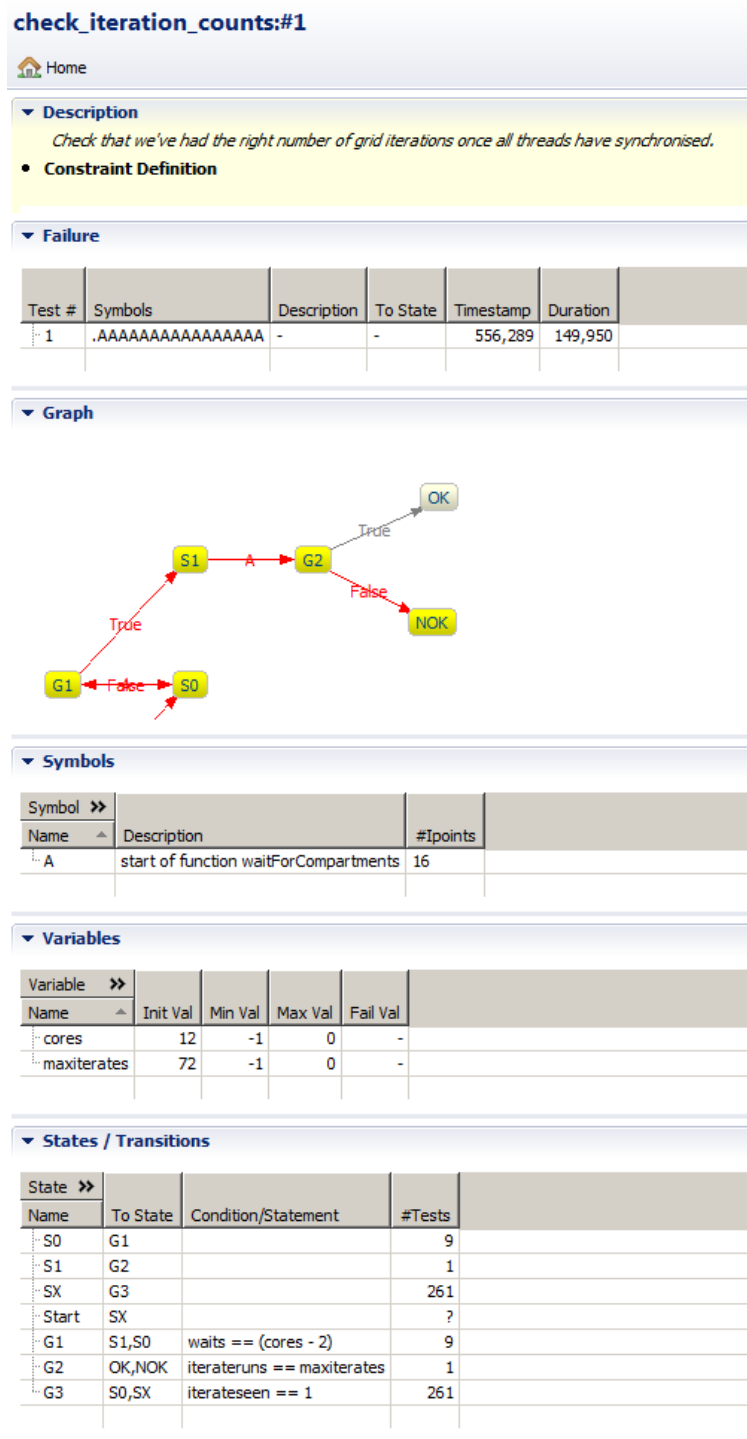


Figure 12: Detailed constraint failure information

2.4 Visualization and profiling tools

Two tools were also developed and/or extended for visualisation and profiling purposes as part of the parMERASA project. For assistance in parallelisation, the Rapita Dependency Tool was developed, and visualisation of the final parallel system was accomplished using RapiTask (which has been extended to support the parallel applications written in the parMERASA project).

2.4.1 Visualisation tool to assist with parallelisation of existing sequential software

Early in the project, we developed a tool designed to assist the Work Package 2 partners in parallelising their sequential code. This tool focused mainly on presenting detailed data, but also offered some visualisation features. The tool itself developed considerably during the first half of the project, but eventually evolved into the Rapita Dependency Tool.

The tool focusses on identifying *dependencies* in code, with dependencies defined as elements of the code which share global variables. Finding this information in a sequential program often produces significant amounts of data, so extra features were added to allow the user to concentrate on specific elements of the program for dependency information. To facilitate this, the tool supports the concept of *aggregates*, allowing the user to select elements of the call tree of interest. This causes the tool to only produce data on global variables shared between those elements of the call tree rather than the entire program.

The tool is able to display some information about dependencies without collecting any traces, as global variable usage can be determined simply from structural analysis of the source code. However, once a trace is parsed, the tool can output access counts for all global variable reads and writes, and will also identify if any elements of the call tree have ‘read after write’ or ‘write after read’ dependencies.

Output is presented in two alternative formats; XML or CIRCOS. The XML output is designed for simply capturing raw data for use in another tool, where the CIRCOS output can be used for visualisation. The CIRCOS format allows the user to feed the information from the tool into the CIRCOS visualisation tool, which presents tabulated data in a circular layout, representing relationships between the data as connections.

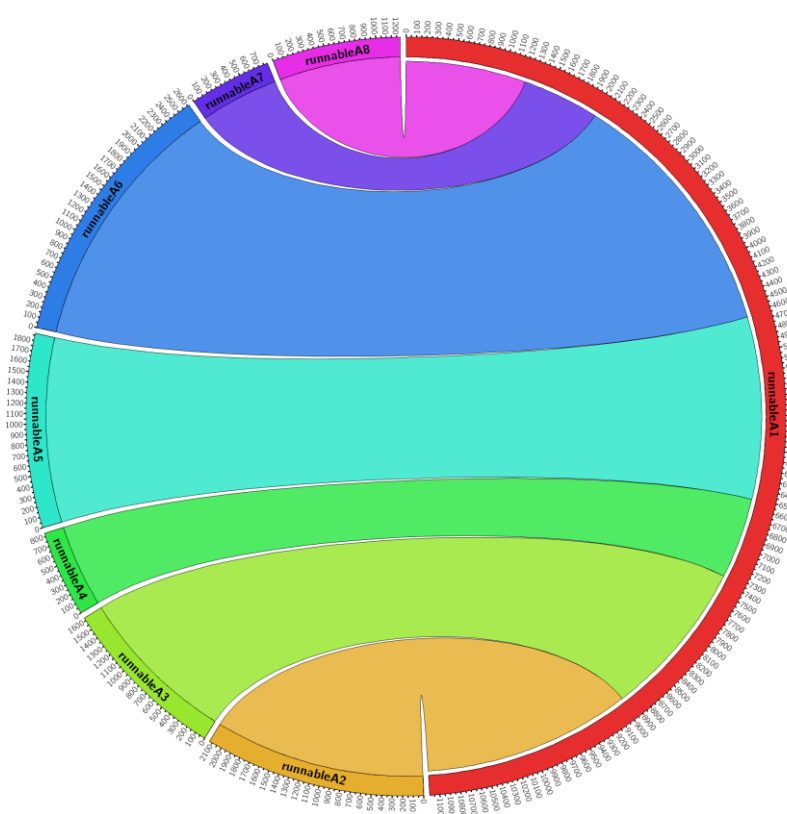


Figure 13: CIRCOS output from Rapita Dependency Tool

Figure 13 shows some example CIRCOS output from an example application containing 8 runnables, with runnables 2-7 consuming data produced by runnable 1 (represented in the XML output as a Read after Write relationship on a global variable).

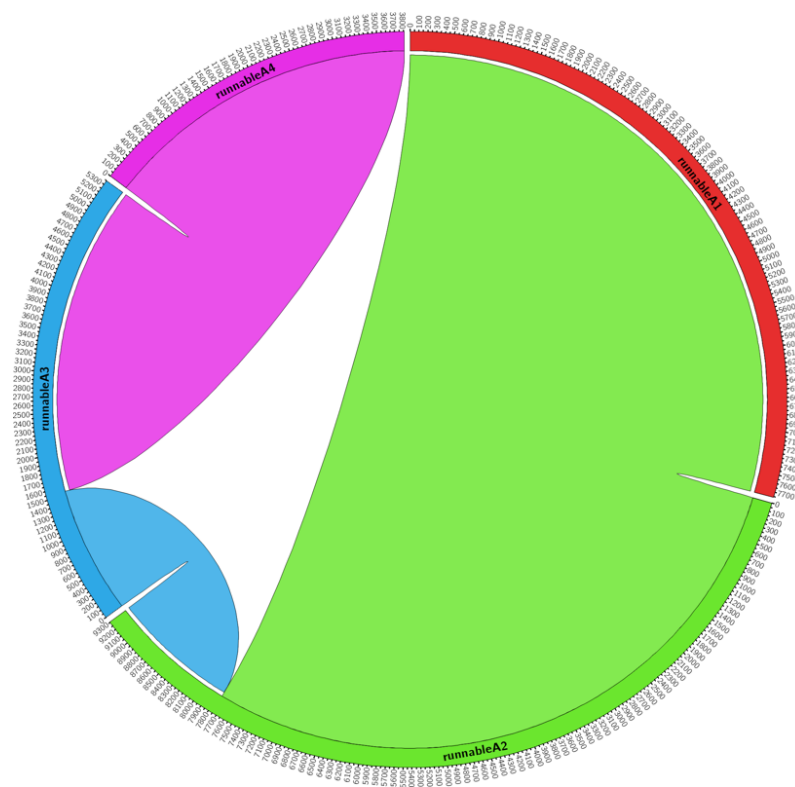


Figure 14: CIRCOS output representing a pipeline

Figure 14 shows an alternative set of example code which makes up a pipeline, with runnable 1 at the beginning and runnable 4 at the end. In the circus output, we can clearly visualise the connections between each runnable (again represented as Read after Write relationships in the XML output).

This tool was successfully used to guide parallelisation of the ‘Dynamic Compaction’ code provided by BAUER Maschinen as part of Work Package 2.

2.4.2 Parallel system visualisation and profiling tool

Visualisation and profiling of parallel code in parMERASA is provided by the RapiTask tool included with the Rapita Verification Suite. RapiTask is a trace visualisation tool, and can be invoked from the RVS Report Viewer.

RapiTask displays the traces collected during the integration progress, and can display information in both task view (displaying only traces from the top level tasks in the system, as defined by the user) or function view (which displays tracing for all functions executed in the trace).

Figure 15 shows RapiTask displaying Honeywell’s 3D Path Planning application in function view. Note that this is a 16 core trace, and as such, the extra cores can be seen by scrolling to the right.

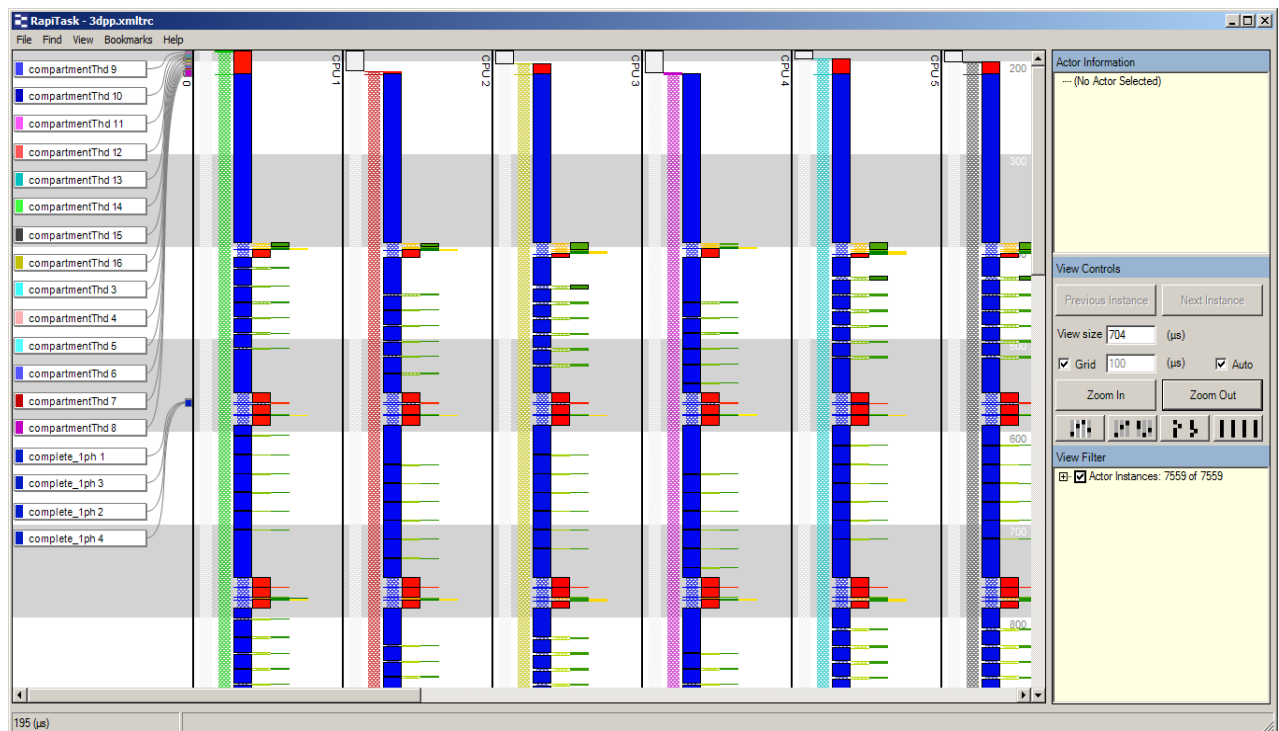


Figure 15: RapiTask displaying 3DPP in Function View

RapiTask has also been extended to support parallel programs as part of the parMERASA project, allowing the display of any number of traces which have executed across multiple cores. The RapiTask Viewer provides information on any selected section of the trace, including start time, end time, response time and execution time, allowing the user to effectively track down poorly performing sections of code and investigate the individual runtimes of all elements of their call tree.

RapiTask has been successfully used on three of the parallelised applications in the parMERASA project, including the 3D Path Planning and Stereo Navigation applications provided by Honeywell and the Dynamic Compaction application provided by BAUER Maschinen.

Visualising the traces of 3D Path Planning is as we would expect. We can clearly see the lockstep data-parallel nature of the application, including points where all threads synchronise at a barrier and portions of the code where each thread works on chunks of work. Figure 16 shows the visualisation of this behaviour.

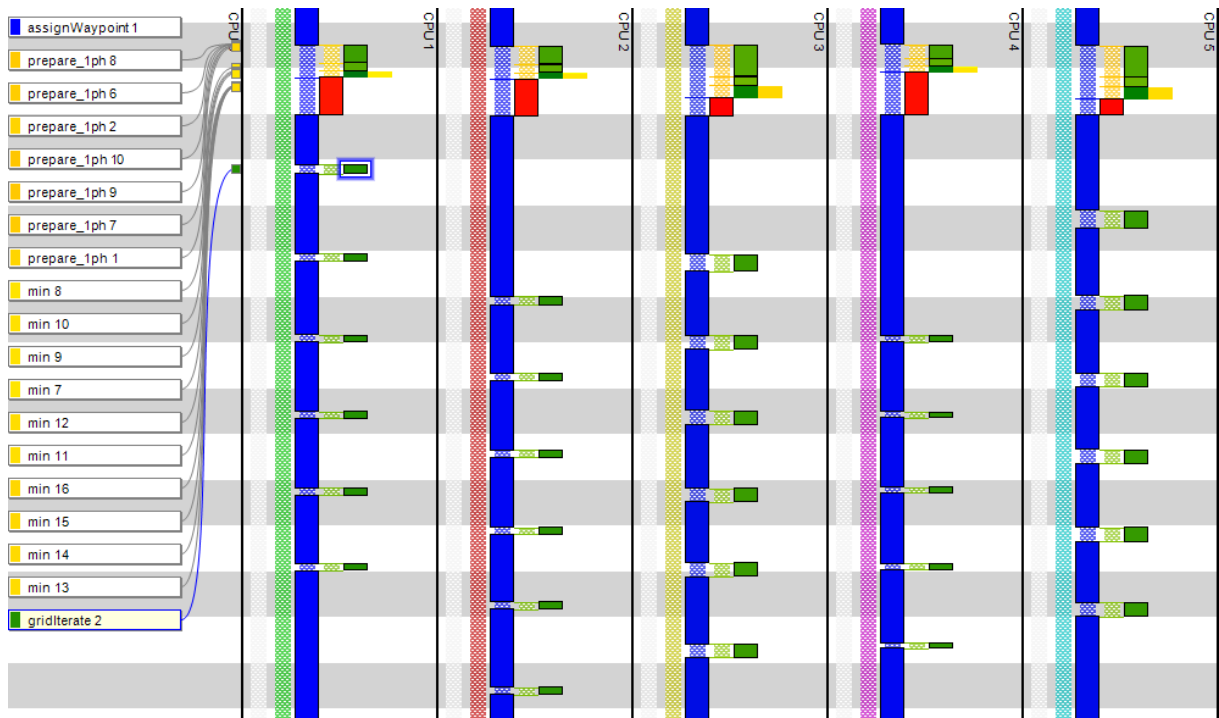


Figure 16: griditerate calls displayed in RapiTask

Analysing the trace with RapiTask also shows interesting patterns of behaviour that often aren't obvious from simply looking at timing information. In this example, we can see that at the beginning of the execution of 3D Path Planning, all but one core is waiting while a single core reads data to prepare for computation. In Figure 17, we can see all but one core waiting at a barrier, while CPU9 conducts a series of functions to read information in from memory (orange/yellow bars).

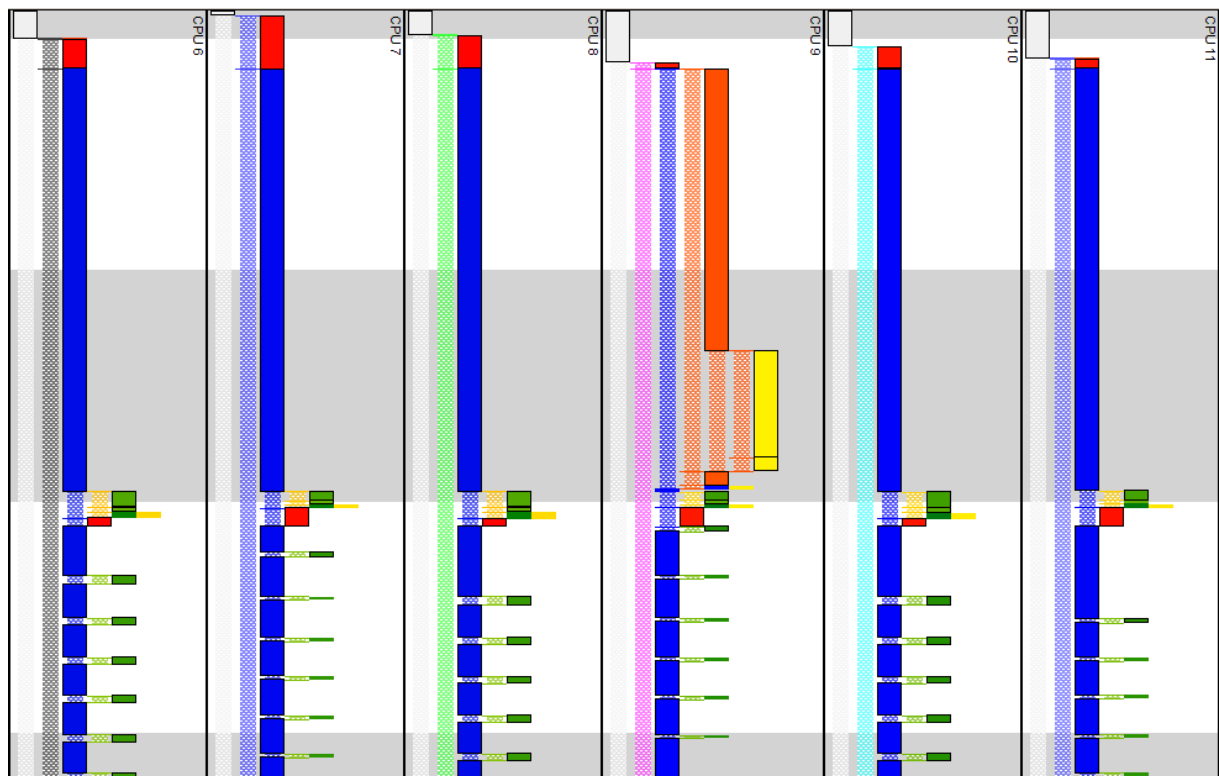


Figure 17: Behaviour of core startup in 3D Path Planning

Insights such as this can be extremely useful for optimising scheduling patterns or looking for better ways to parallelise a given application.

Viewing the Dynamic Compaction software in RapiTask presents a very different trace due to the task parallel nature of this application compared to 3D Path Planning.

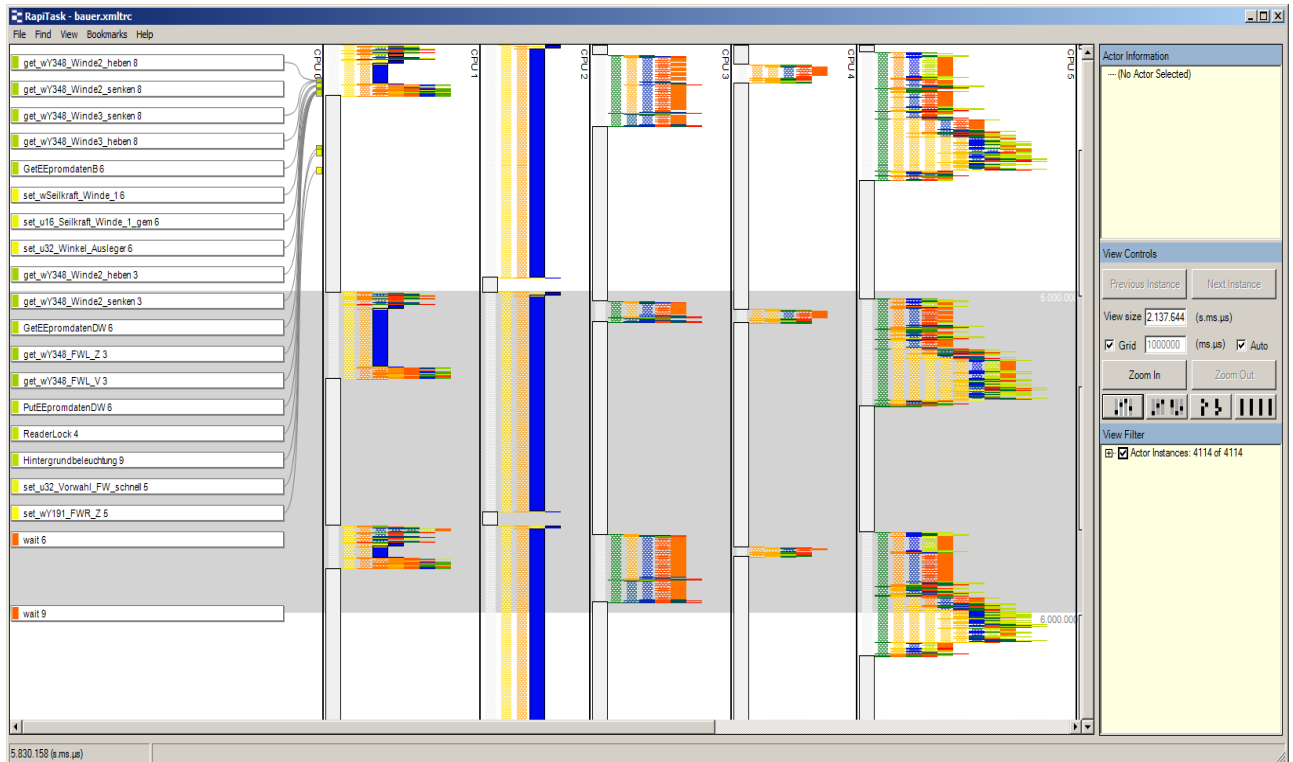


Figure 18: Dynamic Compaction shown in RapiTask

Figure 18 shows a high level overview of BAUER’s dynamic compaction application in RapiTask. We can see, at a glance that the overall parallel behaviour of the application differs to that found in 3D Path Planning, with one core executing what appears to be a single thread of control, while other cores execute parallel tasks with varying degrees of complexity and execution time. Clearly, this application isn’t operating in a data parallel, lockstep way like the Honeywell application. RapiTask can be extremely useful when attempting to get an overall understanding of the pattern of execution inside some software under analysis, and its extension to parallel programs allows developers to easily view and understand the parallel behaviour within their applications.

Before developing RapiTask, we performed a survey of existing trace tools to establish whether the capabilities were provided by any pre-existing tools. The survey report is attached to this document.

3 WCET ANALYSIS OF PILOT STUDIES

The four pilot studies that have been parallelised in Work Package 2 have been analysed using both RapiTime and OTAWA to obtain WCET values for both sequential and parallelised variants. These values have been used to derive values for the worst-case speedup for the parallelised versions, indicating the increase in worst-case performance that results from parallelising the code.

Runtime measurements from the simulator, along with measurements obtained from RapiTime, have also been used to determine the actual runtime performance of both sequential and parallel versions. By comparing the maximum observed runtime value with the calculated worst-case execution time, some idea of the overall pessimism of the worst-case values can be established.

3.1 3D path planning (Honeywell)

Honeywell provided two applications for parallelisation in parMERASA, the first of which is a 3D Path Planning application designed to test parallelisation using data parallel methods. A full description of this application can be found in deliverable 2.6.

3.1.1 Structure of the application and approach to its static WCET analysis (OTAWA)

We analysed the core parallel part of the application, i.e. the 3D multigrid Laplacian equation solver, shown in Figure 19.

The main thread does not perform computations but is instead responsible for orchestrating the work assigned to child threads. Several steps are run, with finer and finer grain. Each step then requires an interpolation phase, to refine the grid, and a computation phase during which the calculations are done. Child threads synchronise with the main thread before beginning each phase/step. Each calculation phase includes a loop that iterates over the grid points. Child threads need to synchronise peer-to-peer, before each iteration, in order to respect data dependencies. This is illustrated in Figure 20 where the 3D grid is split into 4 compartments.

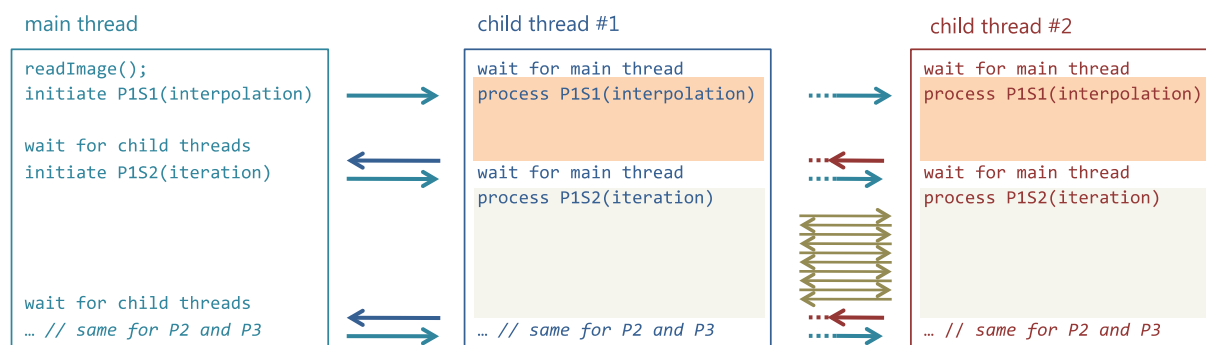
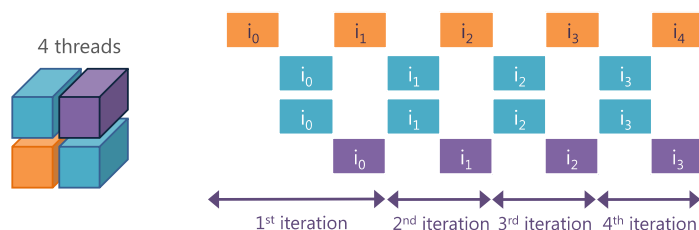


Figure 19: 3D multigrid Laplacian equation solver



3.1.2 Provisional OTAWA analysis and optimisation of the application

UPS has estimated the WCET of this application in cooperation with HON. After preliminary experiments, an iterative optimisation process has been discussed with HON and UAU, and several versions of the code have been provided by UAU.

All the results reported in the following have been obtained considering the parMERASA architecture with a tree-like NoC, 4KB private instruction caches and ODC² caches, and on-chip memory with a latency of 10 cycles.

Initial version – synchronisation with conditional variables

In the initial version of the application, each progress synchronisation was implemented using point-to-point synchronisation with conditional variables (and `signal()` and `wait()` primitives).

Table 1 reports the WCET estimated for three configurations of the code (1-, 4- and 8-thread), as well as the speedup computed from these WCETs.

Table 1: WCET estimates and speedups (condition variables) - 1, 4 and 8 threads

# threads	WCET	WCET-speedup
1	14,078,189	-
4	22,545,686	0.62
8	36,935,432	0.38

It appears that these results do not show any performance improvement with parallelisation (actually, they show performance degradation). We carefully analysed intermediate analysis results and we found out the following issues:

- Application-level synchronisation primitives (`broadcast()`, `wait_for_producers()`, `wait_for_consumers()`) that call `signal()` and `wait()` system-level primitives include critical sections for which maximum contention is assumed. This makes synchronisation costly when the number of threads increases. For the record, the worst-case stall time in `lock()` primitives that guard the critical sections executed in a loop is roughly estimated as:

$$WCST = WCET_{CS} * \#threads * \#calls * \#iterations$$

where $WCET_{CS}$ is the WCET of the critical sections.

- The annotation of synchronisations based on conditional variables is extremely verbose. For each of them, the signalling and waiting threads must be specified, and a complex description should be provided to express that both threads might be in different iterations of the loop. For example, the 4-thread version of the code required 564 lines of annotations, and the 8-thread version required 787 lines.

Second version – synchronisation with barriers

After the analysis of the first version, it was decided to replace conditional variables with barriers. The new implementation was done by UAU in cooperation with HON. The WCET estimates of this version are given below. Now a slight speedup is achieved but it is still far below the expectations.

Table 2: WCET comparison between condition variables and barriers

# threads	with condition variables	with barriers
1	14,078,189	13,540,165
4	22,545,686	11,908,328
8	36,935,432	10,642,793

Table 3: WCET estimates and speedups (barriers) - 1, 4 and 8 threads

# threads	WCET	WCET-speedup
1	13,540,165	-
4	11,908,328	1.13
8	10,642,793	1.27

However, the reduction in annotation needs is noticeable: only 59 lines are required whatever the number of threads (because the annotation of a barrier is the same for any number of threads while point-to-point synchronisations multiply with an increasing number of threads).

Table 4: Annotation comparison between condition variables and barriers - 4 and 8 threads

# threads	with condition variables	with barriers
4	564	59
8	787	59

Third version – with unrolled control loop

Another optimisation of the application came from the observation that the same loop was used to control the calculations for any grid size. Unfortunately, OTAWA cannot account for different flow fact within several executions of the same loop. As a result, it considered the worst-case values (reached for the finest grid) for every step (every grid size). Unrolling the loop was considered as a solution to avoid this and implemented in a third version of the application. As it can be seen in the following tables, this sensibly improved the WCET without affecting the speedup.

Table 5: WCET comparison with and without loop unrolling

# threads	loop	unrolled
1	13,540,165	9,257,327
4	11,908,328	8,266,933
8	10,642,793	7,486,555

Table 6: WCET estimates and speedups (unrolled control loop) - 1, 4 and 8 threads

# threads	WCET	WCET-speedup
1	9,257,327	-
4	8,266,933	1.12
8	7,486,555	1.24

Table 7: Annotation comparison with and without loop unrolling - 4 and 8 threads

# threads	loop	unrolled
4	59	129
8	59	129

3.1.3 Final OTAWA results

The final improvement was reached by implementing a clever address analysis for accesses to arrays. The tables below report the WCETs and the achieved speedup. The speedup value of 1.48 with 4 threads can be seen as a very good result since, as shown in Figure 20, data dependencies limit the maximum theoretical speedup to 2. The speedup achieved with 8 threads is less impressive but still considerable.

Table 8: WCET comparison with and without array analysis

# threads	no array analysis	array analysis
1	9,257,327	7,266,609
4	8,266,933	4,904,399
8	7,486,555	3,954,395

Table 9: WCET estimates and speedups (with array analysis) - 1, 4 and 8 threads

# threads	WCET	WCET-speedup
1	7,266,609	-
4	4,904,399	1.48
8	3,954,395	1.83

3.1.4 Analysis with RapiTime

3D Path Planning was the first application to be analysed using RVS, and was our first major test of the parallelisation features added to the Rapita tools during the parMERASA project. Analysis using RapiTime was only performed on one version of the application, as loop unrolling and call context expansion is supported ‘out of the box’ by RapiTime, and doesn’t require the source of the application under analysis to be altered.

Instrumentation of 3D Path Planning was conducted as described in section 2.1.5, instrumenting and compiling the code using the RVS compiler wrapper.

At this point, it’s worth noting that the analysis of an application using RapiTime supports the use of *annotations* to control the flow of analysis and to provide information to the analysis tools which may aid timing computation. Typically, very few (if any) annotations are required for a successful analysis, but a number of annotations *are* available for certain situations. For example, the *instrumentation profile* is controlled by the following annotation:

```
#pragma RVS default_instrument ("TRUE", "TIME_FULL");
```

This tells the instrument that, unless specified otherwise for a particular function, it should use the ‘TIME_FULL’ instrumentation profile, which inserts enough ipoint calls in the code to collect the data needed to analyse WCET. A number of instrumentation profiles are available, some providing a lower

overhead (in terms of ipoints) but only support code coverage rather than timing, for example. All experiments on the 3D Path Planning application were performed using the 'TIME_FULL' profile, due to the relatively low overhead provided by the ipoint routine in this situation (the addition of a single *nop* per instrumentation point).

Rapitime also has the concept of an *analysis root*. This defines an element of the call tree which should be used as the start point of an analysis. This feature exists for multiple reasons. Often, analysing the entire call tree of an application can consume large amounts of computation or memory, and is often not necessary. The typical use case of RapiTime involves analysing the timing of the computational or time critical elements of an application while ignoring elements such as initialisation code. When analysing 3D Path Planning, after consultation with Honeywell, we chose to use the function *compartmentThd* as the analysis root.

When using the parMERASA extensions for synchronisation primitive timing analysis, the instrument requires the user to specify the name of barrier and/or lock function names. In this instance, we specified to the instrumenter that barriers were called by the function *pthread_barrier_wait*.

3.1.5 Final RapiTime results

Final results for 3D Path Planning were collected from the final version of the application analysed by OTAWA (including manual loop unrolling).

Core Configuration	Computed WCET (Cycles)
1	21822074
2	18117655
4	10213419
8	8395566
16	4768622

More detailed timings (including the maximum observed time from both RapiTime and the parMERASA simulator) can be found in deliverable 2.6.

3.1.6 Comparison of OTAWA and RapiTime results

Clearly the results found by both OTAWA and RapiTime are quite different in the case of Honeywell's 3D Path Planning application, but they both show the same trend of reducing worst case timings as the number of cores increases.

The numbers provided by OTAWA are, overall, much less pessimistic than those provided by RapiTime. We believe there are a few fundamental reasons for the results differing in this way.

Firstly, the two tools and approaches used in this project differ fundamentally in the way they are used. OTAWA, the static analysis tool, requires more input from the developer to create an analysis for any given application. However, if the developer has this amount of time, it has a great deal of options allowing the user to tell the tool fundamental information about the program under scrutiny. While this *may* be time consuming, it can allow for a far tighter WCET analysis at the cost of parMERASA | No. 287519 | D3.12 | Report on support of tools for case studies

complexity of usage. RapiTime, on the other hand, approaches the problem slightly differently, with less complexity and effort required to generate an analysis (with any effort being almost entirely focused on the *integration* stage). However, by decreasing the effort required to output an analysis, RapiTime does not *require* the user to provide hints and optimisation to the analysis tools, causing initial results to often be pessimistic to some degree. RapiTime *does* support a wide variety of annotations which can be used to reduce pessimism, but due to the time required to complete integration on the parMERASA platform, heavy usage of these annotations ended up falling outside the scope of the project.

Secondly, the relative simplicity of 3D Path Planning compared to the other applications in the project allows a static analysis based approach to timing analysis to be particularly effective. The lack of structures which are problematic for static analysis (such as function pointers) allows the majority of effort to be concentrated on tightening timing analysis by re-writing the code to give a better worst case execution time. However, the difficulty and time required to use RapiTime remains constant across all applications due to integration into the platform being the fundamental work needed for any analysis of all three applications.

Ultimately, OTAWA provides more favourable WCET numbers in this case due to the relative simplicity of the application (compared to some of the other applications in the parMERASA project), allowing more time and effort to be directed towards optimisation of the analysis. In the case of RapiTime, the complexity of the application has less of an effect on the time required to generate an analysis, and combined with a simpler and less optimised initial analysis phase, we see more pessimistic timing results.

3.1.7 Evaluation and lessons learned

From RapiTime's perspective, we learned a great deal of lessons from our analysis of 3D Path Planning. It was the analysis of this application which prompted us to overhaul and change the fundamental methodology that RapiTime uses for calculating barrier worst case timing.

Our initial results found when analysing barrier timing were extremely pessimistic compared to both the observed times and OTAWA's estimation. This was due entirely to our method of calculating the timing for any given barrier, as explained in section 2.1.4.

Since this was our first attempt at analysing a parallelised parMERASA application, we also ran into some problems which, we eventually found, were common to all of the applications written as part of the project, including:

- The inability to analyse the calls to barriers and locks which are found inside the system kernel library (see deliverables D4.8 and D4.10). These barriers and locks take the barrier/lock data required for synchronisation in as a parameter, and then call it in a local context. Our barrier/lock analysis has the constraint that barrier and lock calls *must be called on a global variable*. While technically the structures passed are usually global, the obfuscation here can make identification of the variable in use more difficult. It is likely that issues like this could, in future, be solved using automatic resolution at analysis time *or* through the use of annotations.
- A particular compiler optimisation used by gcc caused some issues. When parsing the execution trace, RapiTime always keeps an account of where in the source the trace currently resides by looking at the structural information captured at compile time. If the

trace jumps to a function which, according to the parsed call tree, should not be called at this point in time, the analysis cannot continue. This issue occurred during our analysis of 3D Path Planning due to a compiler optimisation. Even when using `-O0`, gcc will still turn calls to `printf` which contain no format string into calls to `puts`. This causes the execution of the trace to be different to the expected path calculated at compile time. Fortunately, this optimisation can be removed by specifically telling gcc to disable the optimisation.

The problems encountered with this application were issues that occurred across *all* applications analysed in the parMERASA project, and were encountered here first simply because we began our analysis with 3D Path Planning.

3.2 Stereo navigation (Honeywell)

Honeywell's second application is a Stereo Navigation application which has been parallelised as part of the parMERASA project. This application is fully described in Deliverable D2.6.

3.2.1 Analysis with OTAWA: state of progress

For this application, the procedure was the same as for the previous one: WCET analysis by UPS with the help of HON.

Unfortunately, we faced two major difficulties and we could only get preliminary (partial and overestimated) results:

- The first issue is the complex control flow in the application. In particular, parts of the functions are automatically generated code and contain a huge number of loops that cannot be analysed automatically by our tools. As a result, the loop bounds should be annotated manually which we did not have time to do until the end of the project. Additionally, some of them would require assistance from the application designer. Without loop bounds, it is impossible to determine WCETs. Once this problem detected, we have decided with HON to restrict the area of code to be analysed.
- The second issue is that many loads/stores access to addresses that are decided in an initialisation step. Similarly, some loop bounds also depend on the initialisation. Now, the initialisation step is not covered by the WCET analysis. Even if it was, we discovered that initialisations within loops could not be analysed properly by our tools. The same problem was already observed for the 3D-path planning application, for which we developed a specific utility to generate information on the initial state. However, this is tedious, manual work, which we did not have time to reproduce for the Stereo Navigation application. As a consequence, the results we could obtain so far are severely overestimated due to the unpredictable addresses of memory accesses.

3.2.2 Analysis with RapiTime

Honeywell's Stereo Navigation application proved to be *significantly* more difficult to analyse than 3D Path Planning, for multiple reasons, including:

- The time required to run a test of the Stereo Navigation application is *much* longer than 3D Path Planning. While a test of a 16 core configuration of 3D Path Planning takes no more than a few minutes, a test of any configuration of the Stereo Navigation application (based on the final version) takes anywhere between twenty and thirty hours on a comparable host environment. This poses a significant challenge to successfully complete an analysis with RapiTime, as our analysis method requires trace data captured from the application itself

running on the target platform². When coupled with experimental extensions to the toolset, this can cause a serious issue. Occasionally, a bug would be found when attempting to parse the execution traces which required re-instrumentation of the code and regeneration of the traces. Naturally, fast testing and agile bug fixing is almost impossible when trace capture takes more than a day.

- The use of barriers for multi-core synchronisation in Stereo Navigation was found to be incompatible with our update approach used for calculating WCET and WCWT for barriers. As described in section 2.1.4, calculating these timings requires all barrier calls across all cores to be in the same sequence. In the Stereo Navigation application, this was not the case, as some barriers were shared by a subset of threads, and other sets of barriers shared by different subsets. Our initial analysis method would have been able to calculate timings in this instance, but these timings would have been incredibly pessimistic.
- Some elements of the code in Stereo Navigation were written in such a way which made a non-pessimistic analysis extremely difficult. The elements of the code which contribute the most to the worst cast timing of the application contain multiple nested loops, which are very difficult to analyse. However, RapiTime *does* have methods for making the analysis of these kind of structures less pessimistic, but they require re-instrumentation and regeneration of traces, which was extremely problematic in this case due to the long runtime of the application.

3.2.3 Final RapiTime results

Despite the issues described above, we were able to calculate timing information for Stereo Navigation. *However*, these times are likely to be either optimistic *or* pessimistic. This is due to our inability to analyse the synchronisation primitives which were used in the application (as described above).

The result of this is that any part of the code which waits at a synchronisation primitive simply uses the observed waiting time as a basis for a worst case calculation, rather than doing a detailed analysis to calculate a potentially more optimistic or pessimistic one.

Again, like 3D Path Planning, an analysis root needed to be selected. In this instance, our results consider the timing for the function *do_feature_extraction* for parallel codes and *do_feature_extraction_seq* for the sequential implementation.

One other aspect of this application complicates the numbers somewhat. Stereo Navigation implements parallelisation of work into a pipeline. Due to the lack of timing calculation for synchronisation primitives in this case, we see that multiple calls to the analysis root occur, some with drastically higher times than others. This is due to pipeline, with the higher timings being seen at instances of the computation function running at the end of the pipeline. All computation functions begin at the same time, but some wait a significant amount of time before they begin working due to the pipeline fill time. In these results, we have presented the observed and calculated worst case times for all cores in each configuration, with the pipeline fill likely to account for some of the differences in execution time across multiple cores. It is also worth noting that the function *do_feature_extraction* does not execute on every core in a configuration. A more detailed

² The time taken to execute tests is not an issue for static analysis tools such as OTAWA, as it is not necessary to execute the code in order for those tools to perform an analysis.

description of timing calculation and the issues surrounding the pipeline effect can be found in Deliverable 2.6.

3.2.4 Comparison of OTAWA and RapiTime results

Stereo Navigation was a very difficult application to analyse for both Rapitime and OTAWA, but for different reasons. RapiTime struggled due to the time required to execute the application and generate a trace (approximately 30 hours), and OTAWA struggled due to the complexity of the application causing analysis to be problematic. Unfortunately, the complexity meant that no results could be generated with OTAWA.

Rapitime was able to generate results for Stereo Navigation, but they are more pessimistic than we would like, mainly due to the way barriers are used in the application. Barriers in 3D Path Planning are called in the same sequence across all threads (as work is done in lockstep), allowing a detailed barrier timing analysis to take place, considering the work of all other threads at barrier synchronisation points, allowing pessimism of barrier waiting times to be reduced. In Stereo Navigation, barriers were used in a more complex way, causing the sequence of barriers to differ across threads, making a full barrier timing analysis prohibitively complicated. This leads to the timings generated being pessimistic.

In conclusion, both Rapitime and OTAWA struggled analysing Stereo Navigation, but the fundamental differences in complexity of use slightly favoured RapiTime in this instance. With RapiTime's complexity being relatively constant across all applications (unlike the complexity of OTAWA differing with the complexity of the application), RapiTime was able to generate some results (albeit pessimistic ones), where OTAWA had trouble doing so.

3.2.5 Evaluation and lessons learned

From Rapita's perspective, we learned a great deal from our analysis of Stereo Navigation. Firstly, that simulation of complex parallelised applications, even in the embedded space, can take *significant* amounts of time, causing huge problems for our trace based analysis approach, particularly when requiring a fast turnaround time for optimising our timing analysis. Should industry take this approach when porting legacy applications to multi-cores in future, we may need to investigate some alternative methods to test annotation improvements without fully re-instrumenting and re-tracing the application.

We also discovered a particularly problematic use case for our timing analysis of synchronisation primitives, in that the usage of different barriers for different subsets of cores complicates our analysis significantly. We are confident that this issue can be overcome *without* introducing the significant pessimism of our original method, but this would require time and a considerable amount of effort. However, the usage of this methodology in parMERASA suggests to us that this could be a common parallelisation pattern in industry in the future, so our method of achieving parallel timing analysis may have to be improved.

3.3 Diesel engine controller (DENSO)

DENSO Automotive provided a diesel engine controller application to the parMERASA project. As part of their contribution to work package 2, they have parallelised this application. The diesel engine controller is described in detail in deliverable 2.6.

3.3.1 Analysis with OTAWA

For this application, the WCET analysis has been fully performed by the user (DENSO). We have provided them with an operational version of the OTAWA tool and explained the way it should be used during a short visit. DENSO has performed many analyses in the last months and we kept in contact to help them solve one-off difficulties. We also added a few functionalities to the tool on request. For example, we added the ability of determining the WCET of a function call on the worst-case path, possibly considering several call sites.

WCET estimates computed by DENSO are provided in Deliverable D2.6.

3.3.2 Analysis with RapiTime

Much like our partners in Toulouse, the RapiTime analysis of the DENSO Diesel engine controller has been conducted by DENSO themselves, with aid provided by Rapita Systems where required. This is largely due to restrictions on sharing of source code. However, we have been given regular status updates and results during the course of the project, as RapiTime supports sending result files with source code stripped, containing only the captured timing results and function names (but no source code).

3.3.3 Final RapiTime results

The final results of the analysis of the Diesel engine controller will be published by DENSO as part of deliverable 2.6.

3.3.4 Comparison of OTAWA and RapiTime results

Since all analysis results of the DENSO Diesel engine controller application have been collected in-house by DENSO themselves, a comparison of the results achieved using OTAWA and RapiTime can be found in Deliverable 2.6.

3.3.5 Evaluation and lessons learned

Working with DENSO has provided Rapita with some interesting feedback, albeit very different feedback to the lessons learnt analysing the other applications in parMERASA. In particular, our inability to analyse the code directly has allowed for a significant amount of input on how the use of our tools by users outside the company could be improved. Not only this, but some of the requirements of DENSO in this project has led directly to improvements in our parMERASA related projects, including:

- The development of the dependency tool, and in particular, its aggregation function, was designed to aid DENSO in their parallelisation of AUTOSAR runnables.
- A renewed focus on RapiCheck and parallelisation equivalence checking for automotive was based on feedback directly from DENSO.
- Improved integration methods and compiler wrappers are currently under development based partly on feedback received from the users of RapiTime at DENSO.

3.4 Dynamic compaction (Bauer)

BAUER Maschinen have parallelised an application which performs ‘dynamic compaction’ using a large crawler crane. Full details of this application can be found in deliverable 2.6.

3.4.1 Structure of the application and approach to its static WCET analysis

The application consists of a set of tasks and it has been parallelised using the *task parallelism* pattern. This is illustrated in Figure 21.

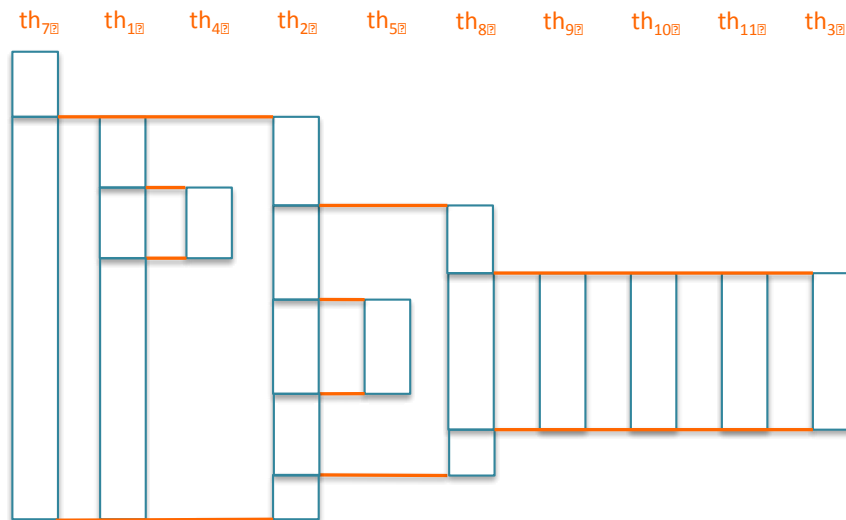


Figure 21: Task parallelism in the 10-thread version

A number of threads are created in the initialisation part. Then thread 7 (the *main* thread) is in charge of sharing work with some of the other threads (threads 1 and 2). To do so, it assigns a function pointer to a specific variable and meets the selected thread at a synchronisation barrier. In turn, some of the threads that have been assigned work by the main thread share it with other threads (e.g. thread 2 assigns work to thread 5). Once shared work has been completed, the involved threads meet at a synchronisation point. The schedule shown on the figure is repeated in an indefinite loop. What has been computed is the WCET of one iteration of this loop.

Threads share variables that are accessed within separate critical sections protected by locks (a single shared variable is manipulated in a critical section).

UAU provided us with:

- a description of the application structure, expressed in an XML format
- annotations for synchronisations which are required to perform the WCET analysis of a parallel program. They include information on threads possibly competing for critical sections (which threads, parts of code executed within the critical section) as well as on threads meeting at synchronisation barriers (which threads, previous collective synchronisation point). Such annotations are expressed in a dedicated XML format that has been designed in WP3 and was described in Deliverable D3.2.

Currently, the WCET analysis cannot be performed fully automatically due to the generic code executed by child threads that dynamically gets work through a shared variable that is assigned with a function pointer. This behaviour cannot be detected from the binary code. It should be manually annotated and developing support for such annotations is still on-going work. The results provided in this report have been obtained by analysing the functions individually, then by combining their WCETs taking the overall application structure into account. Nevertheless, the impact of

synchronisations on the WCET of functions is automatically accounted for thanks to the available annotations.

3.4.2 Analysis with OTAWA

The original application was sequential code. It has been parallelised by UAU and BMA as part of WP2, using parallelisation patterns. Several exchanges between UPS and UAU have led to a parallel version that is analysable by the OTAWA toolset.

While analysing this application, we faced several difficulties:

- A major issue with this application is the size of its code. To achieve accuracy, static WCET analysis in OTAWA virtually inlines functions when it builds the control flow graph (CFG). This allows a contextual analysis of the WCETs of functions, i.e. an analysis that takes the call site into account. Due to the large number of function calls, such virtual inlining generates millions of virtual basic blocks which requires a large memory capacity: we estimated that analysing one task of this application would require more than 200 GB of memory. To tackle this problem, we implemented a tool that estimates the weight of each function (in terms of instructions count and number of calls). We then extended OTAWA to allow the user to specify whether some functions (the heaviest ones) should not be inlined. However, guaranteeing a safe WCET analysis with such an extension still requires work on the OTAWA code, which is clearly out of the scope of this project. Additionally, we found out that the timing analysis of partially-inlined code is still very long: even though we decided to split tasks in several chunks of code, we need several days to estimate the WCET of one chunk. Finally, the manual work needed to combine the WCETs of chunks and tasks is significantly long and error-prone.
- As for the HON Stereo Navigation application, a tight timing analysis would require a preliminary analysis of the initial memory state. Unfortunately, such an analysis cannot be performed automatically and requires manual assistance. Due to the further difficulties reported below, we failed to do it before the end of the project.
- Another difficulty is due to the presence of large switch statements in the code. The problem is that the way such statements are compiled is very much dependent on the compiler. As a result, it is complex to develop a safe analysis that automatically discovers the possible targets of the indirect branch used to implement the switch statement. We decided to annotate those targets manually, but this is very tedious and error-prone (in particular for statements with several hundreds of cases).

For these reasons, we could currently obtain results only for a limited number of configurations, and those results are overestimated.

3.4.3 Analysis with RapiTime

The dynamic compaction application provided a number of challenges when it came to analysis using RapiTime. While we were able to achieve a full analysis of the code, it was time consuming and involved multiple hurdles. Some of the major issues we encountered with the application were:

- The parallelisation approach used in the dynamic compaction application presented some difficulties for our analysis tool. Ultimately, the parallelisation method is a very effective and sensible one from a software development point of view, unfortunately, the architecture of our tool and our methodology for analysing applications meant that analysis of software

parallelised in the style of this application proves tricky. For example, the use of function pointers to dispatch worker tasks to the cores means extra time is required at analysis time to resolve function pointers to their targets based on trace data.

- The concepts of analysis roots means that a full understanding of call trees for each worker running on each core needs to be understood, otherwise traces cannot be de-multiplexed effectively. The concept of *de-multiplexing* traces is crucial to a RapiTime integration which contains multiple tasks or threads. Typically, a single core outputs a single trace. If the application contains multiple tasks that pre-empt each other (or even run sequentially), the trace will contain output from both tasks interlaced together. When parsing the trace, this is problematic, as we can see the trace jump from one element of the call tree to one which is totally disconnected (due to another task being scheduled in). In this instance, we must *de-multiplex* the traces to separate the traces from each task into their own trace file. This is easy to accomplish, *provided there is a good understanding of which tasks run on each core*. This was problematic in the BAUER application, and changes based on the amount of cores in the system, meaning a single integration cannot be used for all configurations easily.
- The code for the dynamic compaction software is by far the largest code base in the parMERASA project. While the time needed to run the code isn't huge (if a relatively small number of loops around the main loop are performed), the time needed to complete the analysis of the code is significantly higher than the other applications in the project, with a full analysis taking around 45 minutes compared to 5 minutes for 3D Path Planning and about 10 – 15 minutes for Stereo Navigation (not including trace collection time).
- The build system for the dynamic compaction application is complicated. While this won't cause a problem for an OTAWA analysis (since they work on the generated binary), a complicated build system can be exceeding problematic when attempting a RapiTime integration. Thankfully, our usage of a compiler wrapper is relatively un-intrusive, and any issues with the build system were fixed collaboratively with BAUER.

3.4.4 Final RapiTime results

A basic overview of the final results achieved with RapiTime is as follows:

Core Configuration	Maximum Observed Cycles	Calculated WCET (Cycles)
1	991,109	994,127
3	1,102,258	1,122,715
4	1,006,124	3,127,630
8	675,045	4,586,565
12	720,044	3,589,194
16	710,194	4,341,522

These numbers only consider the main algorithmic skeletons of the application, and a more detailed explanation of the numbers can be found in deliverable 2.6.

3.4.5 Comparison of OTAWA and RapiTime result

BAUER's dynamic compaction code was very difficult to analyse using either OTAWA or RapiTime. The code itself is possibly the most complex application developed in the parMERASA project. Where the other applications are pilot studies or smaller examples of their respective systems, BAUER's parallelised code was based on the full codebase used in a production model of the dynamic compaction system.

OTAWA were able to generate some numbers for the system, but the numbers generated represent a subset of the system and are pessimistic. RapiTime was able to generate a full set of results for the system, but this required significant effort and merits some discussion and justification.

Firstly, the fundamental way that RapiTime constructs results and analyses timing clashed with the method used to parallelise the BAUER application (the algorithmic skeletons). This made a full calculation of timing for the entire application difficult, but not impossible. However, to achieve this, some of the numbers had to be composed together by hand with knowledge of the source code and scheduling parameters to create overall worst case and max observed times.

Thankfully, the use of barriers and locks in the BAUER application fit correctly with RapiTime's multi-core analysis method, allowing a fully considered generation of timing information taking the behaviour of all cores into account.

An interesting observation found in both the RapiTime results and the preliminary OTAWA results are a high amount of pessimism compared to the maximum observed execution times. However, looking at the full results collected using RapiTime, we believe that the majority of this pessimism is justified for two reasons:

- The system itself is, fundamentally, a mixed criticality system, but the timing evaluation undertaken considered the entire system as a whole rather than focussing specifically on the safety critical elements of the code. If time allowed, we likely would have continued to focus the analysis further on elements of the code which considered timing critical. However, the results obtained were thorough enough to identify that the pessimistic timings are often due to high worst cases found in non-critical elements of the system.
- The method used to compose timings to create the analysis was not entirely comprehensive. Due to the way RapiTime works, we currently don't support the automatic composition of timing results which are dispatched across multiple cores in a complicated fashion. Due to this, we needed to work with BAUER to manually compose the timing collected based on how the system was scheduled.

3.4.6 Evaluation and lessons learned

Thanks to the use of clear parallelisation patterns, the Dynamic Compaction application did not raise any major issue for static WCET analysis. The only difficulty that was encountered is due to the use of generic code for statically-generated child threads. Such code makes use of function pointers, which would require an additional level of analysis to determine which code is eventually executed by each thread. This could not be implemented before the writing of this report but we do not foresee specific problems that would prevent us from doing it short-term.

From Rapita's point of view, some *extremely* important lessons were learned from our work analysing the dynamic compaction application. In our view, the most important lesson learned here regards the usage of sensible parallelisation patterns and how they interact with our fundamental parMERASA | No. 287519 | D3.12 | Report on support of tools for case studies

analysis methodology. It's evident that the parallelisation approach taken by BAUER makes a great deal of sense from a software architecture point of view. It appears to work well for parallelising a fundamentally task parallel application, and makes a great deal of sense when used with legacy applications. Unfortunately, this approach increased the difficulty of generating a full code analysis using RVS. The methodologies we use for analysing single core applications are finely tuned and work effectively in the single core case, but can be problematic when analysing a multi-core application. We found that data parallel applications which run in lock-step with no pre-emption don't cause problems for our method, but task parallel applications with multiple tasks running on each core can be difficult. Due to this, it's likely that we will revisit and change some of the underlying technology behind our toolset in future as more industrial embedded applications begin to consider multi-core execution.

A second major taking from working with the dynamic compaction code involves our methods for *integrating* RVS with a target build system. The integration stage of using RVS has historically been the most difficult part of the analysis process, and our experience with the BAUER codebase was no exception. Working with complicated build systems often leads to multiple cascading build failures if even small changes or errors occur, and unfortunately, it's often very difficult to integrate RVS into a build system without making some small changes to the customer's build system, especially when multi-core code is involved. Some of the problems encountered during our integration of RVS into the dynamic compaction build system are now being investigated as part of a general overhaul of our integration process.

Overall, we consider our work with BAUER to be a success, as we've been able to successfully analyse what is arguably the most complicated parallelised application in the parMERASA project, generating meaningful results which support the parallelisation method used in the dynamic compaction application.

4 CONCLUSIONS ON THE SUPPORT OF TOOLS FOR CASE STUDIES

Ultimately, we consider the work undertaken in work package 3 (development of tools for case studies) to have been a success. While the results collected are often more pessimistic than we may have wished, a significant number of challenges arose as part of the project, and we believe that collecting a comprehensive set of results, even though they are pessimistic, is a significant achievement. In our view, we consider the following points to be significant successes:

- Effective support for analysing the timings of multi-core programming constructs, displayed through the analysis of all the parallelised applications created in the parMERASA project.
- Extension of a number of verification tools to allow their usage in development of embedded systems.
- Development of new tools which are fundamentally designed to aid in parallelising software.
- Successful integration of a ‘hybrid style’ timing analysis tool with a complicated experimental simulation platform.
- Significant additions to a ‘static style’ timing analysis tool for multi-core support, including very positive results indicating the feasibility of static analysis for multi-core applications.
- Effective usage of trace viewing and constraint checking verification tools on industrial parallelised applications.

Reviewers identified that a comparison between existing trace visualisation tools and the proposed extensions to RapiTask for parMERASA be completed. This report can be found attached to this deliverable.

REFERENCES

- [1] parMERASA consortium, “Deliverable 2.6 - Final evaluation results on parallel industrial applications,” 2014.
- [2] parMERASA consortium, “Deliverable 3.1 - Requirements on WCET analysis and verification tools,” 2012.
- [3] parMERASA consortium, “Deliverable 3.2 - Specification of tool support for the analysis of parallel applications and guidelines for program annotations,” 2013.
- [4] Rapita Systems Ltd., “RapiTime,” [Online]. Available: <http://www.rapitasystems.com/products/RapiTime>.
- [5] C. Ballabriga, H. Cassé, C. Rochange and P. Sainrat, “OTAWA: an Open Toolbox for Adaptive WCET Analysis,” in *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, October 2010.
- [6] M. De Michiel, A. Bonenfant, H. Cassé and P. Sainrat, “Static loop bound analysis of C programs based on flow analysis and abstract interpretation,” in *Int’l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2008.

LIST OF FIGURES

Figure 1: Analysis of parallel programs	19
Figure 2: Observed execution time, estimated WCET and pessimism.....	20
Figure 3: 3D multigrid Laplacian equation solver.....	33
Figure 4: Data dependencies	34
Figure 5: Task parallelism	43

Multi-Core Execution Tracing Tools

David George

This report outlines the state of execution tracing tools designed for multi-core systems as of October 2012. Rapita currently develops an execution tracing tool designed for single core embedded systems. Unfortunately this tool does not support concurrent programming environments. This report aims to investigate available options for multi-core tracing with a view to either extending the Rapita suite of tools or exploring alternative options. Ultimately, a multi-core tracing tool is required for the parMERASA project, and this document aims to make a series of suggestions about how this deliverable might be achieved.

Disclaimer

The information, text and graphics contained in this document are provided for information purposes only by Rapita Systems Ltd. Rapita Systems Ltd. does not warrant the accuracy, or completeness of the information, text, and other items contained in this document.

Confidentiality

The information contained in this document is confidential and may also be proprietary and trade secret. Without prior approval in writing from Rapita Systems Ltd. no part of this document may be reproduced or transmitted in any form or by any means, including but not limited to electronic, mechanical, photocopying or recording or stored in any form of retrieval system whatsoever. Use of any copyright notice does not imply unrestricted public access to any part of this document.

Copyright and Trade Marks

All rights reserved. Information and images contained within this document are copyright and the property of Rapita Systems Ltd. All trademarks are hereby acknowledged to be the properties of their respective owners.

Contacts

For further information please contact
enquiries@rapitasystems.com
www.rapitasystems.com

Rapita Systems Ltd
IT Centre
York Science Park
Heslington
York
YO10 5DG

Registered in the UK number 5011090

Document Control

Title: Report Title
Revised: 25/10/10
Version: 1.0
Part: DOC-20080125

1. Initial Concerns

While investigating multi-core execution tracing tools a number of fundamental issues surrounding the viability of particular tool-sets became apparent. These issues are important to consider when investigating tools that may be appropriate for the parMERASA project.

1.1. Online vs. Offline Analysis

The first obvious dichotomy between tracing tools involves the decision to use offline (or post-mortem) analysis of an execution trace or online analysis based on repeated execution and measurement/instrumentation at runtime. Most tools considered in this investigation concentrate on analysis of traces *post* execution (such as Vampir, RapiTrace or Paraver), though others use online analysis (Periscope), and the methods that other tools use to analyse traces is either unclear or appears to be a hybrid approach (Pajé). Since, in all likelihood, any trace viewer provided by Rapita will be closely tied to RapiTime and the remainder of the RapiTime Verification Suite, online performance analysis for multi-core application traces is likely to be of limited value, as the rich instrumentation provided by RVS coupled with the relatively low performance of embedded platforms (compared to high performance clusters) lends itself more favourably to a post-mortem approach to tracing and analysis.

1.2. Tracing File Formats

Trace viewers (be they designed for single or multi-core execution environments) ultimately exist to view information from program execution in a manner that facilitates performance optimisation. Any trace viewer that uses offline (or post-mortem) analysis of an execution trace will need to load information about the execution from a *trace file*. Multiple competing formats exist within the community for trace files, including:

- Open Trace Format
- Slog2 (Jumpshot)
- TAU format
- Pajé format
- RapiTrace format
- Paraver format

Most of these formats are specific to their own ecosystems (e.g. the TAU format is created by and utilised by the TAU instrumentation and trace viewing tools), but converters do exist to turn some formats into others. The Open Trace Format is also becoming popular across multiple toolsets, with multiple tracing and instrumentation tools now offering output in OTF format allowing usage in alternative trace viewers.

1.3. Instrumentation vs. Analysis

Most multi-core tracing tools are not simply considered with analysis of tracing data but are also interested in how data from an execution trace can be captured. Some tools have a mixed approach offering detailed instrumentation *and* analysis, while other tools focus on achieving one or the other. For example, the Tau analysis tools are more concerned with the instrumentation of programs rather than their eventual analysis. On the other hand, tools such as Jumpshot and Pajé are more concerned with graphically visualising traces instead of providing a set of detailed instrumentation tools.

1.4. Levels of Granularity

A key requirement for a tracing tool focusing on embedded and real-time platforms is a level of analysis granularity that makes sense for the platform. The majority of existing multi-core tracing and analysis tools are primarily designed for High Performance Computing platforms (which have significantly different visualisation demands than an application running on an embedded multi-core processor). Figure 1 shows some of the analysis features provided by Vampir. These views are focused more on the *overall* bottlenecks of the system as a whole based on the activities of the processes within it. A tool designed to focus on embedded systems is likely to be more useful when focusing on Worst Case Execution Time metrics and individual blocks of code running within a process. The approach taken by Vampir makes sense on a system of a massive scale (such as those used for HPC projects), but the granularity of information analysed will need to be smaller for embedded multi-core systems. This will allow software developers to focus on individual lines of code that may be causing problematic WCET values. Users of HPC platforms are more interested in achieving better average case execution across the entire platform (usually caused by inefficient concurrent algorithm implementation or communication bottlenecks) where embedded system designs are seeking to find particular lines of code which contribute to a higher *worst case* execution time. The granularity of trace visualisation needs to directly reflect these alternative goals.

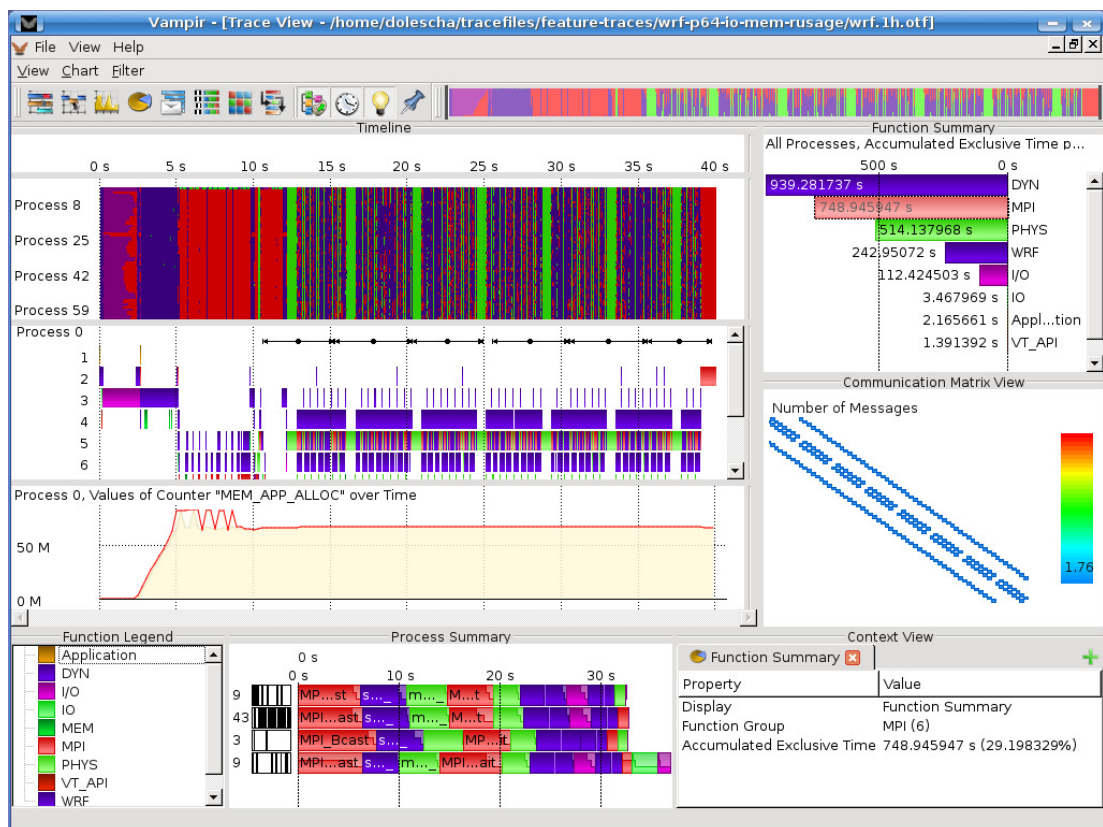


Figure 1 - Analysing an MPI based execution trace with Vampir.

1.5. Source vs. Binary Instrumentation

Another key difference in the tools presented here is their approach towards instrumentation of applications. While online vs. offline analysis has already been discussed, there still remains one fundamental difference in how applications can be instrumented when conducting offline/post-mortem analysis in the form of source or binary instrumentation techniques. Some analysis tools prefer to insert extra code into the application source to create instrumentation points and trace data, while others may offer instrumentation of applications *without* their associated source code. Instrumentation of the application then takes place by injecting instructions into the binary or using measurements from alternative sources (such as memory usage, processor usage and communication library calls). Our likely approach to a trace tool would *not* involve using third party instrumentation libraries due to the rich set of features provided by RapiTime and its measurement and instrumentation functionality. This also means that trace viewers designed with binary instrumentation in mind may be inappropriate for use with RapiTime instrumentation. Specifically, it is unlikely that a trace viewer paired with an instrumentation library utilising binary injection will ever have source code available to use in trace analysis. Analysis of a trace and its associated source code is highly likely to be an integral aspect of a tracing tool for embedded multi-core applications (due to an increased focus on WCET metrics), making a trace viewer without the ability to tie trace events to source code blocks inappropriate for our needs.

1.6. Parallel Library Requirements

Many existing multi-core tracing tools are focused on HPC applications. This is simply due to the historic need of parallelisation and concurrent programming to achieve the maximum possible performance available across machines of huge scale (such as clusters and supercomputers). However, multi-core platforms are now becoming the norm across the entire computing space, leading real-time system designers to examine how their systems could potentially benefit from concurrent execution. With respect to trace viewers, their focus on HPC concurrency paradigms could cause difficulty in their usage in an embedded context. For some time, concurrent programs in the HPC space have utilised relatively heavyweight communication libraries (such as MPI, OpenMP and PVM) to manage communication between concurrently executing threads. These paradigms are unlikely to be used in the embedded world due to their size and complexity. Unfortunately, this presents an issue when considering the adaptation of existing trace viewers for use in an embedded context. Many trace viewers targeting multi-core execution focus on displaying metrics from parallel programming APIs (such as MPI) as a methodology for analysing performance of parallel programs. This approach may not translate well to programs that use alternative methods for parallelisation (such as shared memory based communication and locking or lightweight thread based parallel libraries).

2. Existing Tracing Tools

This section outlines a number of execution tracing tools currently available and evaluates their suitability for use in the parMERASA project. These tools are not necessarily all built for multi-core tracing, but are included because they are either readily available or have the potential to be altered to fit our requirements.

2.1. Paraver

Paraver is Barcelona Supercomputing Centre's multi-core performance analysis toolset, comprising both instrumentation and measurement tools alongside an advanced visualisation library allowing detailed analysis of multi-core traces.

Key points:

- Traces can be generated using multiple high level concurrent programming libraries including MPI, OpenMP and Java2EE.
- Paraver can instrument code and create trace files through source modification or alternative methods using binaries only.
- The toolset provides a fairly large set of visualisation tools for trace analysis, including communication tracing (Figure 2), execution timelines and a powerful 'semantic module' allowing advanced filtering of data in an execution trace based on various criteria.
- Paraver provides multiple tools to generate traces from a variety of formats, providing multiple paths to analysing traces from a variety of systems.

On the whole, Paraver appears to be a reasonably complete toolkit for analysing parallel programs, but there is some concern over whether these tools would be immediately suitable for analysing traces from embedded multi-core systems. The level of abstraction in trace analysis seems to be too high for our usage, especially considering the lack of source code views and a focus on high-level HPC concurrency libraries as metrics for performance analysis. Ultimately, Paraver appears to be a well-rounded tool for analysing medium to large scale programs in the HPC space, but would likely need moderate modification to allow its usage in the embedded industry.

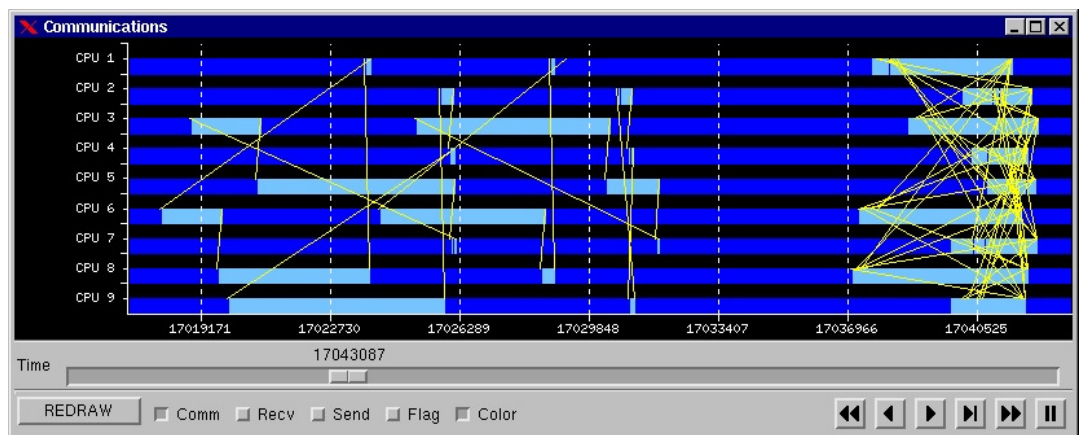


Figure 2 - Paraver MPI communication tracing.

2.2. Vampir

Vampir is a commercial multi-core performance analysis toolset developed from work at TU Dresden. Vampir has a very clearly segmented architecture composed of a trace viewer (Vampir), tools for instrumentation and tracing of programs (VampirTrace) and server software for analysing large traces using clusters (VampirServer).

Key points:

- VampirTrace uses source modification to achieve instrumentation by wrapping compilation in special Vampir specific scripts.
- The toolset is evidently extremely concerned with open standards, as the provided instrumentation tools are also part of the OpenMPI standard and traces comply to the Open Trace Format.
- Large trace files are easily managed using VampirServer.
- Vampir isn't constrained to displaying traces that are created by VampirTrace. Due to the conformance to open standards, many other instrumentation tools can convert their traces to OTF, allowing multiple traces to be viewed and analysed in Vampir.
- Vampir unfortunately suffers from similar issues to other tools presented here, in that the granularity of its information display is too large for use in embedded tools. The visualisation tools are most concerned with showing high-level performance metrics (such as communication calls, memory usage, cache misses and scheduling information) rather than per statement tracing, though Vampir does include call tree analysis.
- Vampir is very easy to build and use, and presents a slick interface which has clearly been well tested and developed.

Vampir is evidently one of the more accomplished and well-produced sets of tools for performance analysis of HPC programs. Unfortunately, this doesn't necessarily translate to meaningful trace analysis of real-time multi-core programs. While traces are easy to convert into formats supported by Vampir, the high level of abstraction is still an issue. However, it's entirely possible that some modification to include statement-by-statement performance analysis *could* be possible, though whether this is feasible given the commercial status of the product would require further analysis.

2.3. Tau

Tau (Tuning and Analysis Utilities) is another set of tools designed for HPC multi-core performance analysis originating from the University of Oregon. Tau has a strong focus on the instrumentation and measurement side of tracing rather than viewing and analysis of data.

Key points:

- Tau is able to generate trace data in a number of formats, including OTF (for Vampir), SLOG2 (for use with Jumpshot) and the Paraver trace format.
- The toolkit supports multiple different parallel programming paradigms (including MPI and OpenMP).
- Tau supports user-defined events, allowing application developers to have more control over information exposed in their traces.
- Despite the focus on instrumentation, Tau does contain some visualisation tools

that offer basic visualisation and reporting along with some oddly outlandish viewing methods (including three dimensional graphing of trace data - See Figure 3).

- Tau allows binary-based instrumentation in the absence of source code.

Tau is a well-established tool for multi-core performance analysis, but it evidently fits into a very specific niche, specifically that of instrumentation and measurement, which it provides a large number of features for. While visualisation tools are provided, this toolset is ultimately designed for instrumentation and not analysis, making it relatively useless to us.

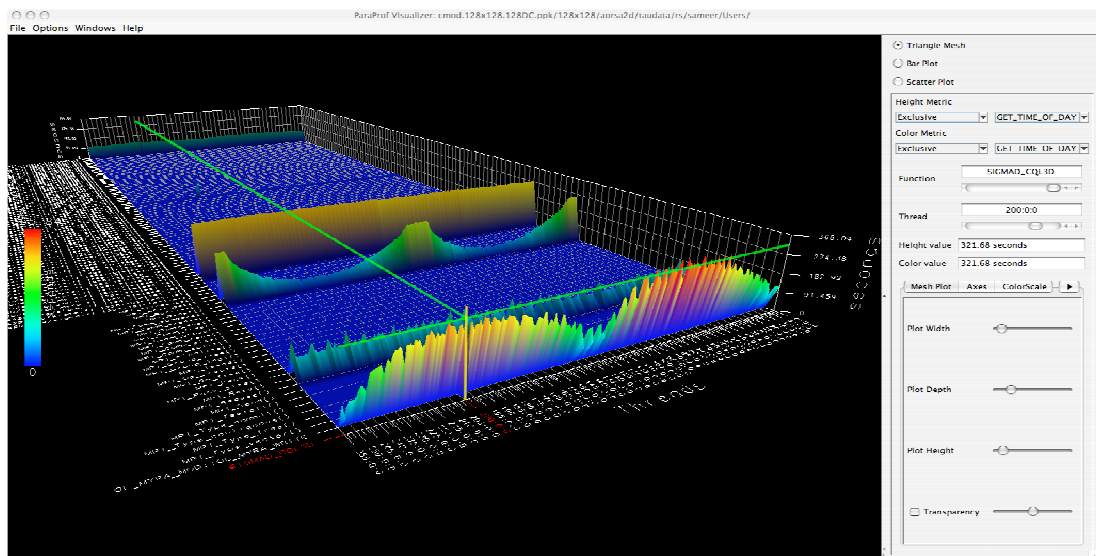


Figure 3 - Tau's 'ParaProf' visualisation tool.

2.4. Open|SpeedShop

Open|SpeedShop is a multi-core tracing tool developed as a community effort (in conjunction with the Krell Institute) and is aimed mainly at Linux clusters and HPC applications.

Key points:

- OSS is capable of instrumentation at runtime using a tool called 'Dyninst'. This tool is capable of injecting instrumentation points into a binary at runtime.
- Crucially, OSS is capable of line-by-line analysis of code with timing information gained from dynamic instrumentation. Aside from this feature, OSS has relatively few tools for trace analysis.
- Ultimately, OSS is a performance analysis tool aimed at high performance scientific codes. It uses instrumentation from a variety of sources, including PC data, clock time, floating-point exceptions, hardware counters and MPI calls.
- OSS is also capable of outputting traces to OTF format.
- OSS supports tracing parallel programs using libraries other than MPI, though it seems that parallel code analysis is not necessarily the primary goal of this tool.

OSS is an interesting tool due its somewhat unique focus on dynamic instrumentation using

binary injection. This tool also appears to be focused on Linux applications and scientific codes distributed across clusters. Unfortunately, while the tool contains line-by-line timing analysis, it doesn't contain many other tools for visualisation and data analysis, making it relatively unsuitable for our needs.

2.5. LTTng

LTTng (Linux Trace Toolkit Next Generation) is one of the main tracing toolkits used and advocated by the Linux community, and as such, is as an open source project. The main goal of LTTng is to replace sporadic and user specific tracing techniques, providing a unified toolset for tracing multi-core programs running in Linux environments.

Key points of LTTng:

- LTTng focuses on tracing single threaded programs rather than concurrent applications.
- The toolkit is able to trace both kernel space and user space applications running on Linux systems.
- LTTng is ultimately more concerned with instrumentation of applications rather than displaying trace data.
- Despite the lack of multi-core visualisation features, LTTng allows other developers to extend the system using plugins.

LTTng appears to be a useful set of tracing tools for the Linux community, but unfortunately it is lacking many of the features that are desirable for our usage (Specifically the lack of a focus on concurrent programs and a focus on instrumentation over analysis). The lack of support for alternative platforms other than Linux (due to the very nature of the projects aims) is also problematic. However, the extendibility of the toolkits visualisation functions may be useful in the future (provided alternative OS environments don't require support).

2.6. Jumpshot

Jumpshot is the latest in a long line of trace visualisation tools originating from the Argonne National Laboratories. Jumpshot originally grew out of the 'Upshot' and 'Nupshot' tools, with the added 'J' standing for 'Java', added due to the implementation of a Java based GUI for the most recent version of the tool.

Key points:

- Jumpshot is primarily concerned with trace visualisation and analysis rather than with instrumentation and measurement.
- The tool parses and displays execution data from files in the SLOG-2 trace format. There appears to be some support from alternative tool vendors (such as Tau) for outputting files into SLOG2 format. While the format isn't open, it is well defined.
- The tool appears to support a rich set of visualisation options, allowing those viewing the traces to easily see a set of high-level information about a trace (see Figure 4). Unfortunately, the versions investigated still have a reasonably high level of granularity and are focused on HPC based parallel programming APIs.
- The team developing Jumpshot have a very clear set of goals for their visualisation toolset which line up very well with the requirements for a tool that will be appropriate for the parMERASA project. Specific requirements which fit with our needs include:
 - "Support for MPI concepts, such as communicators, is required. At the

same time, it is overly restrictive to tie the tool to the message-passing model of parallel computation.”

- “It is desirable that an event can be connected back to the source code that generated it.”
- “The tool must be portable to all the types of workstations that parallel programmers would want to use. Although the use of X made previous tools portable within the Unix universe, portability to Microsoft-based environments is now required.”
- “The display package (the Jumpshot part) must not be too tightly integrated with the logging package (the CLOG part), for the sake of flexibility within each.”

While these are stated as goals for the project, it isn’t immediately clear to what extent these goals have been achieved in recent versions of the toolset. This would require further investigation.

Jumpshot appears to be an extremely promising candidate for analysis of multi-core traces in the parMERASA project. While the tool is currently focused on HPC applications, the aims of the tool authors align closely with our requirements, and further investigation into whether the tool could be appropriate either ‘out of the box’ or via some modification would be useful.

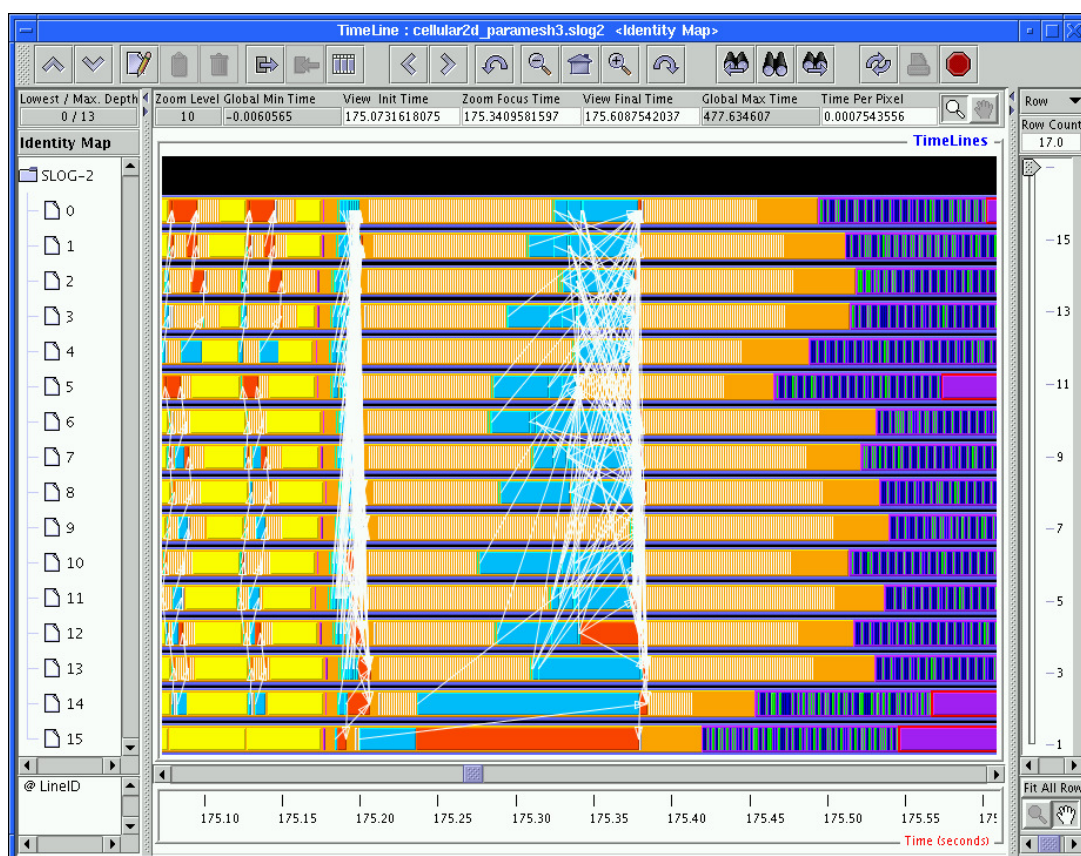


Figure 4 - Timeline based communication and execution visualisation using Jumpshot 4.

2.7. Periscope

Periscope differs from the other tools presented here in that its primary goal is online performance tracing and analysis. This tool is currently developed at TU Munich.

Key points:

- Like most tools presented here, Periscope is focused on HPC performance analysis rather than embedded or real-time tracing.
- The main goal of Periscope is similar to that of LTTng, in that the authors aim to replace the outdated methodology of chaining together multiple GNU tools when creating a tracing and analysis platform.
- The key difference between Periscope and alternative tools is the use of online analysis and *automatic discovery* of potential performance bottlenecks. Periscope uses a multi-agent framework to conduct iterative analysis of the program under scrutiny while attempting to dynamically discover possible areas for further analysis.

Like a number of other tools presented in this document, Periscope is more concerned with the instrumentation and measurement side of performance analysis and tracing. While the methods used to measure and identify performance bottlenecks are novel, they don't offer anything meaningful to the parMERASA project (due to our pre-existing instrumentation framework). Despite this, there are things that *can* be learned from the Periscope approach. While it may not be useful for us to use the product in its entirety, attempting to use some of the approaches that the TU Munich team have used in Periscope with our own instrumentation and analysis tools could prove fruitful.

2.8. Scalasca

The Scalasca toolset (developed at the Jülich Supercomputing Centre) is another HPC focused tracing and performance analysis tool that aims itself at clusters and supercomputers running massively parallel programs.

Key points:

- Scalasca, like many other tools, has instrumentation as its main focus. The apparent goal of the toolset is the increased performance of instrumentation and tracing across massively parallel clusters.
- This tool is able to instrument parallel communication data from MPI, OpenMP or 'hybrid' libraries, though also supports the concept of user defined events, allowing users to have more control over the information collected in their traces.
- Like most other tools presented here, the level of abstraction is higher than might be expected in an embedded environment. This tool is very clearly aimed at the HPC industry.
- Scalasca also contains line-by-line source code analysis, a feature that is desirable in an embedded multi-core tracing tool.
- Despite being strongly focused on instrumentation, Scalasca contains a fair amount of trace analysis and reporting tools. Unfortunately, these tools are aimed firmly at HPC users, mainly considering performance metrics that are of interest to developers working on hugely parallel software.

Scalasca is clearly focused at HPC developers. Ultimately, despite some promising features (such as line-by-line source code analysis based on traces), this tool is unlikely to be appropriate for usage in the parMERASA project. However, I believe that some of the innovations presented in Scalasca may be worth reviewing in the future. In particular, the optimisations and tweaks that the developers have made while attempting to better scale tracing across massively parallel systems may be worth investigating should our

instrumentation tools have issues scaling to many-core designs.

2.9. RapiTrace

RapiTrace is the real-time/embedded tracing tool sold by Rapita Systems. This tool is a rebadged and tweaked version of Tracealyzer, a tracing tool developed by Percepio.

Key points:

- RapiTrace acts as a trace viewer alone and contains no code to take measurements or instrument running code. As part of the RapiTime Verification Suite, traces can be collected using RapiTime then viewed in the RapiTrace trace viewer.
- RapiTrace doesn't currently support the analysis of multi-core traces, and focuses on the analysis of traces from a single core to identify bottlenecks and stalls in real-time applications.
- Ultimately the tool is relatively simplistic, focusing on a small amount of metrics important to real-time developers (such as WCET and response time). Despite this, the granularity of information displayed is relatively high, and tracing events cannot be directly tied to line-by-line analysis of source code.
- This tool contains a moderate amount of support for advanced reporting, but unfortunately this support pales in comparison to some of the alternative HPC focused tools (such as Vampir, Jumpshot or Paraver).

Unfortunately, RapiTrace isn't suitable for analysing multi-core execution traces in its current state. However, the foundations exist for creating a more advanced trace viewer. The GUI is polished and effective at showing the limited information required for a single core trace, but whether the GUI will continue to be effective in the face of larger scale multi-core data is unclear.

2.10. Pajé

Pajé, like Jumpshot, is another tracing tool that focuses almost entirely on the visualisation of tracing data rather than measurement and instrumentation. The Pajé toolset grew from the work of the Apache research project into a community directed open source tool.

Key points:

- Pajé aims to analyse programs that create a dynamic number of threads rather than a fixed amount (like HPC focused code using parallel libraries such as MPI). This aligns much better with the goals of parMERASA, as this methodology better matches the parallel programming models of real-time/embedded systems.
- This tool also focuses on analysing parallel programs which use *shared memory based communication* rather than heavyweight message-passing libraries (such as MPI).
- Pajé's GUI is designed to be extensible, allowing developers to create new ways of interpreting and examining trace data.
- Pajé's visualisation tools contain some important features that bring the level of viewing abstraction to a level that is far more appropriate than most of the other tools presented here. These include:
 - Line-by-line statement viewing based on data recovered from a trace.
 - Visualisation of shared memory based locking between processes (see Figure 5).
 - Analysis of semaphore based locks (see Figure 5).

Unfortunately there appears to be some ambiguity surrounding how Pajé handles trace data when conducting analysis. The tool claims to be a post-mortem style trace viewer, yet there seems to be some discussion of 'simulation' when moving around the trace. This needs further investigation, as a tool that conducts dynamic instrumentation while viewing code would be inappropriate for use with our existing measurement and instrumentation tools. Despite this, Pajé looks to be an extremely promising tool. It very closely matches our goals for an embedded/real-time multi-core tracing tool, but one outstanding issue remains: the trace file format. A number of tools exist to convert traces to Pajé format, but investigation will need to be conducted to discover whether RVS could be capable of outputting traces to a format supported by Pajé. However, despite these issues, Pajé appears to be a very good candidate for adoption as a multi-core embedded trace file viewer.

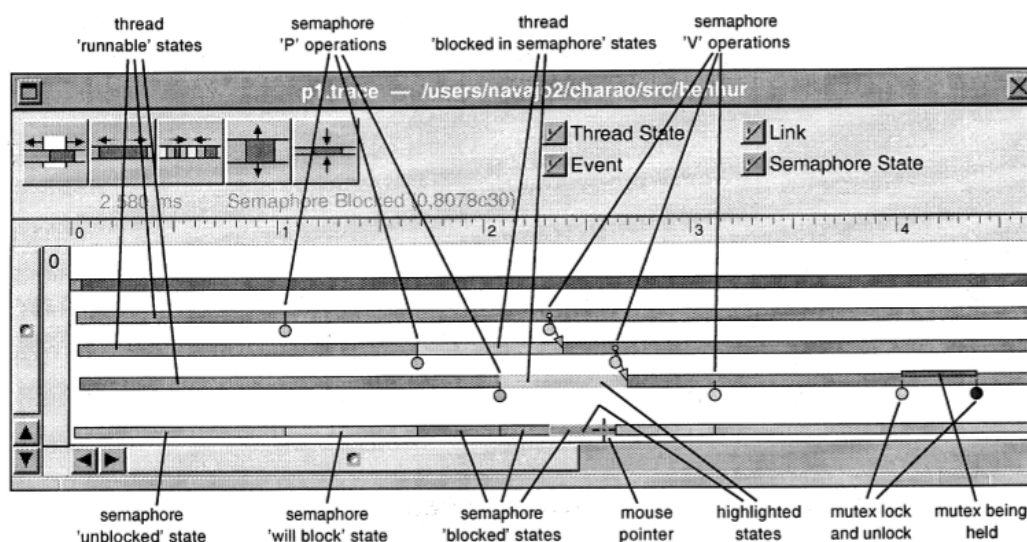


Figure 5 - Multi-thread execution analysis using Pajé.

2.11. Other Tools

A number of other tools exist that haven't been considered in this document, either due to their being out-dated, having inappropriate licensing or due a simple lack of time.

These tools are:

- Pablo
- Traceview
- XMPI
- XPVM
- Microsoft HPC Toolkit
- Sun Performance Analyser
- ParaGraph

3. Conclusions

The tools presented in this document represent a high level overview of the main multi-core trace analysis platforms current available. The aim of this work is to discover which tools are worth considering for further analysis, ultimately creating a deliverable multi-core tracing tool for the parMERASA project. Unfortunately, the majority of tools share similar issues, specifically:

- Many tools are designed with High Performance Computing in mind, relying on heavyweight parallelisation libraries (such as MPI or OpenMP).
- Most tools present data in a form that isn't entirely appropriate for analysing embedded multi-core platforms. Displaying small granularities of data (such as fine-grained locking and line-by-line source analysis) will be essential for a real-time/embedded focused multi-core tracing tool.
- Some tools may have licensing issues that prevent ease of modification or redistribution.
- Using the existing instrumentation functionality of RVS is highly likely to be part of our overall strategy for multi-core tracing in parMERASA. This precludes many of the tools presented here due to their focus on instrumentation and measurement over display and analysis.

Based on these factors and our requirements, two tools stand out as good candidates for further investigation. Jumpshot, the Java based trace analysis tool, while focused on HPC applications, may be adaptable for use in an embedded environment. Pajé also appears to be worth extra investigation. The level of abstraction provided in Pajé is much more appropriate for real-time and embedded applications than HPC focused tools, and while there are some outstanding issues, further investigation into both of these tools is recommended.