Universität
Augsburg
University

University of Augsburg
Department of Computer Science
Systems and Networking

Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores

PhD thesis

Dissertation for the degree of
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

Author:
Dipl.-Inf. Mike Gerdes

Supervisor:
Prof. Dr. Theo Ungerer
2nd Supervisor:
Prof. Dr.-Ing. Rudolf Knorr

Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores

Thesis committee:

Supervisor/Examiner: Prof. Dr. Theo Ungerer

2nd Supervisor/Examiner: Prof. Dr.-Ing. Rudolf Knorr

Examiner: Prof. Dr. Bernhard Bauer

Day of defense: July 17, 2013

Nearly every man who develops an idea works it up to the point where it looks impossible, and then he gets discouraged. That's not the place to become discouraged. — Thomas A. Edison

Success is 1 % inspiration, 98 % perspiration,
and 2 % attention to detail. — Phil Dunphy

Abstract

The thesis on hand provides hardware-software co-design of timing analysable synchronisation techniques in embedded shared-memory multi-core processors. In hardware, an augmented memory controller including the logic to support consistent and atomic Read-Modify-Write (RMW) primitives for a predictable shared-memory multi-core processor has been developed. The techniques introduced with the augmented memory controller are also applicable to further (future) shared-memory multi-core platforms. On top of these RMW primitives, timing analysable software synchronisation techniques are provided. On the one hand, hard real-time (HRT) capable, worst-case efficient, lock-based synchronisation techniques employing busy-waiting (spinning) and blocking (suspending) are introduced. On the other hand, worst-case efficient barrier implementations for progress coordination of HRT threads are presented.

The implemented hardware and software techniques are analysed in detail with an open-source static worst-case execution time (WCET) analysis tool, which supports the analysis of multithreaded parallel programs on shared-memory multi-cores. The static timing analysis includes worst-case memory latencies for the concurrent access of threads to a shared global memory, and allows for analysing multithreaded programs by introducing worst-case waiting times at synchronisation points. Furthermore, two benchmarking parallel programs, a parallel matrix multiplication and an Integer Fast Fourier Transformation, have been implemented and analysed using the proposed synchronisation techniques.

The analyses have shown that busy-waiting spin locks are preferable over locking techniques with suspension for multithreaded parallel programs on shared-memory multi-core processors. Also, the static timing analyses indicate that pessimism in the WCET estimates could be further reduced by providing a technique that prioritises frequent normal load and store operations over infrequent RMW operations on synchronisation variables. The optimisation technique, the split-phase synchronisation technique, has been implemented in the augmented memory controller, and allows for splitting RMW operations into a load and modification phase and a store phase while maintaining atomicity and data consistency under a weak consistency model. The WCET analyses of parallel benchmark programs with the split-phase synchronisation technique applied shows an additional improvement of WCET guarantees.

The remainder of this thesis introduces an approach towards a novel parallelisation method for HRT programs using parallel design patterns. The pattern-based parallelisation process integrates the developed HRT capable synchronisation techniques as synchronisation idioms. The proposed parallelisation approach is envisioned to be further evolved and integrated in a software engineering approach that limits the possible variability in the parallelisation process to well-known, timing analysable structures by embracing programmer and timing analyser knowledge in forms of design patterns, algorithmic skeletons, and idioms.

Zusammenfassung

Die vorliegende Dissertation behandelt Hardware-Software-Co-Design von zeitlich vorhersagbaren Synchronisierungstechniken für eingebettete Mehrkernprozessoren mit gemeinsamen Speicher. Dazu wurde der Speicher-Controller eines echtzeitfähigen eingebetteten Mehrkernprozessors um die Hardware-Logik zur Behandlung von atomaren und konsistenten Read-Modify-Write (RMW) Primitiven erweitert. Die entwickelten Techniken lassen sich auch in weiteren (zukünftigen) Mehrkernprozessoren verwenden. Darauf aufbauend werden zeitlich analysierbare, worst-case effiziente Software-Synchronisationstechniken bereitgestellt. Einerseits werden Synchronisationstechniken mit Busy-Waiting (Spinning) und Sperren (Aussetzung) und andererseits Barrieren zur Fortschrittskoordination von HRT (harte Echtzeit) Kontrollfäden vorgestellt.

Die implementierten Hardware- und Software-Techniken werden im Detail mit einem open-source, statischen worst-case execution time (WCET) Analyse-Tool, das die Analyse von mehrfädigen parallelen Programmen auf Mehrkernprozessoren mit gemeinsamen Speicher unterstützt, evaluiert. Das benutzte WCET Analyse-Tool verwendet worst-case Speicherlatenzen zur Behandlung von konkurrierenden Zugriffen mehrerer Programmfäden auf einen gemeinsamen Hauptspeicher. Es ermöglicht die Analyse von mehrfädigen Programmen durch die Einführung von worst-case Wartezeiten an Synchronisationspunkten. Außerdem wurden zwei parallele Benchmark-Programme, eine parallele Matrixmultiplikation und eine mehrfädige (*Ganzzahl*) Fast Fourier Transformation, entwickelt und unter Verwendung der vorgeschlagenen Synchronisationstechniken evaluiert.

Die Evaluierungen ergeben, dass für die echtzeitfähige Ausführung von mehrfädigen Programmen auf Mehrkernprozessoren mit gemeinsamen Speicher Busy-Waiting Spinlocks gegenüber den Synchronisationstechniken, die einen aktiven Kontrollfaden aussetzen, vorzuziehen sind. Außerdem zeigen die vorgenommenen Echtzeitanalysen, dass der Pessimismus in den WCET Abschätzungen weiter reduziert werden kann, wenn eine Technik eingesetzt wird, die sehr häufige normale Lade- und Schreibzugriffe gegenüber selteneren RMW Operationen auf Synchronisationsvariablen priorisiert. Die Optimierungstechnik ist im erweiterten Speichercontroller implementiert und erlaubt das Aufspalten von RMW Operationen unter Aufrechterhaltung der Atomarität und Datenkonsistenz unter einem schwachen Konsistenz-Modell. Die erzielten Ergebnisse mit der Optimierungstechnik zeigen eine weitere Verbesserung der WCET Garantien.

Am Schluß dieser Dissertation wird ein Ansatz für eine neuartige Methode zur Parallelisierung von HRT-Programmen mit parallelen Entwurfsmuster vorgestellt. Dieses Parallelisierungsverfahren beinhaltet die entwickelten HRT-fähigen Synchronisierungstechniken als Synchronisations-Idiome. Das vorgeschlagene Parallelisierungsverfahren wird zukünftig weiterentwickelt um die mögliche Variabilität des Parallelisierungsprozesses noch weiter auf bekannte, zeitlich analysierbare Strukturen zu begrenzen. Dazu wird Wissen von Software-Entwicklern und Echtzeit-Analysten in Form von Entwurfsmustern, algorithmischen Programmskeletten und Idiomen gesammelt und aufgeführt.

Danksagung

In erster Linie bedanke ich mich bei Prof. Dr. Theo Ungerer und Prof. Dr.-Ing. Rudolf Knorr für die Begutachtung dieser Dissertation.

Besonders möchte ich mich bei Prof. Dr. Theo Ungerer für seine hervorragende Betreuung während meiner Promotionsphase bedanken. Seine zahlreichen Hinweise, seine konstruktive Kritik und seine Tipps zum Inhalt und Aufbau der Dissertation haben wesentlich dazu beigetragen, dass die Arbeit in dieser Form entstehen konnte. Außerdem möchte ich mich herzlich bedanken für die Möglichkeit an diversen Tagungen, Konferenzen und Workshops teilnehmen zu können. Insbesondere die aktive Teilnahme an zwei EU-Projekten hat nicht nur zum Gelingen und Vertiefen der Inhalte dieser Arbeit beigetragen, sondern auch zu einer äusserst angenehmen und fruchtbaren Zusammenarbeit mit den Kollegen der Université Paul Sabatier in Toulouse geführt.

Je voudrais me remercier encore une fois particulièrement à Christine Rochange (et les autres à l'IRIT, Université Paul Sabatier, Toulouse). Les discussions intensives et la coopération étroite dans les projets et les publications ont m'a beaucoup aidé et contribué à la réussite de la thèse.

I am also very grateful for the funding received from the European Union-funded FP7-projects MERASA (under grant agreement no. 216415) and parMERASA (under grant agreement no. 287519).

Auch möchte ich allen Kollegen in Augsburg und am Lehrstuhl herzlichst danken. Sei es für die zahlreichen kritischen Diskussionen, auch über mein eigenes Themengebiet hinaus, die Zusammenarbeit bei Publikationen und die gemeinsamen Dienstreisen (insbesondere auch vielen Dank an Petra, die immer hilfreich zur Seite stand), die gemeinschaftlichen Mittagessen (insbesondere am Fast-Food-Friday), oder auch die zahlreichen Aktivitäten außerhalb der Arbeitszeit (Badminton, Hardware-Bier, Lehrstuhl-Plärrer etc.). Ebenso danke ich auch allen Studenten, deren Abschluss- und Projektarbeiten mir sehr bei der zeitnahen Implementierung und beim Ausprobieren verschiedener Ideen geholfen haben.

Vielen Dank gilt auch allen Freunden (v.a. denen, die ich leider aufgrund der Entfernung nur noch sehr selten zu Gesicht bekomme: Nadi&Flo, Biggi, Zummi, Kerstin&Ingo), die für die nötige Ablenkung und Unterstützung außerhalb der Arbeitszeit gesorgt haben. Insbesondere möchte ich mich auch noch bei den Freunden bedanken (Maria, Lexa&Thomas, Julia&Roberto), die während meiner Dienstreisen so liebevoll auf Krümel aufgepasst haben.

Ich danke auch besonders meiner Familie für ihre Unterstützung, die mir auch ermöglicht haben ohne große finanzielle Sorgen nochmal ein spätes Studium aufzunehmen. Danke auch für die vielen Care-Pakete, die mich, trotz der großen Entfernung, regelmäßig mit Süßigkeiten (und Socken) versorgt haben.

Augsburg im April 2013 ⌣̈ Mike Gerdes

Table of Contents

Time is a great teacher, but unfortunately it kills all its pupils. – Hector Berlioz

1 Introduction

Multi-core processors have been introduced to the market in around 2005 when single-core processors reached their power limit on frequency scaling. Since then, they successfully dominated the domain of general purpose computing and also largely poured into the embedded space. Today, nearly every new smartphone on the market utilises at least a dual- or even quad-core processor. Also, the automotive and avionic industry is highly interested and researches the possible use of multi-core processors in real-time and cyber-physical systems, because of their promising performance/watt ratio and the possible capability of quenching their thirst of an increasing demand in higher performance.

In real-time computing, timing requirements and correctness, additional to functional correctness, need to be satisfied. Hard real-time (HRT) programs and systems violating their timing correctness can expose catastrophic consequences to users, people or the environment (cf. Buttazzo 2004, Stankovic and Ramamritham 1990). Therefore, systems and programs in that category must verify their timing correctness with the help of intensive testing and/or timing analysis tools. Especially static timing analysis tools (see a survey on tools and methods by Wilhelm et al. 2008) aim at deriving safe upper bounds for the execution, even under the worst possible circumstances, so-called worst-case execution time (WCET) guarantees, taking program flow and microarchitectural structures into account.

Also, spatial and temporal isolation is one of the major design principle for hardware and software architectures in the domain of HRT computing. Main reason are the needs and requirements for functional safety that must be satisfied for certification of such systems in the industry. That is, for example, the developed software must be certified against the domain-specific certifications standards: ISO 26262 in the automotive domain, EN 50128 for railway systems, IEC 61513 for nuclear power plants, IEC 62061 in the machinery domain (all based on the IEC 61508 standard), and the DO-178B standard in the avionic domain. These standards define e.g. criticality levels spanning from low-criticality to the highest criticality. Thus, if isolation between programs, which can be categorised into different criticality levels, cannot be assured, these programs need to be certified against the highest criticality level they share (cf. Baruah et al. 2010).

However, rigorous isolation cannot be fully retained in multi-core processors: not all resources that are needed to execute one or more HRT programs are duplicated in a multi-core processor. That is as the main design idea of multi-core processors versus multiprocessors is that they employ individual processing elements that are tightly coupled through sharing interconnects and memory structures as much as possible. Hence, to execute HRT programs, singlethreaded or multithreaded, on such multi-core processors, one need to find ways to safely upper bound accesses to resources, which are shared or introduce interference and waiting times. Hence, static timing analysis tools for parallel HRT programs on multi-core processors need to face the challenge of analysing and acknowledging such interferences and waiting times from thread-level competition and synchronisation on shared resources (see Brandenburg et al. 2008, Chattopadhyay et al. 2012, Gerdes et al. 2012b, Gustavsson et al. 2010, Rochange 2011, Wolf et al. 2010a).

With the advent of multi-core processors in the real-time embedded domain, different trends have sparked from there: on the one hand the research on deterministic execution of real-time programs on commercial off-the-shelf (COTS) multi-core processors (e.g. Boniol et al. 2012, d'Ausbourg et al. 2012, Hardy et al. 2009, Nowotsch and Paulitsch 2012) and Real-Time Operating System (RTOS) support using virtualisation or hypervisors (see e.g. LynxSecure, Wind River Hypervisor). On the other hand, research in academia explores ways on designing and building timing predictable multi-core processors (cf. Cullmann et al. 2010, Edwards and Lee 2007, Hansson et al. 2009, Liu et al. 2010, Pitter and Schoeberl 2010, Ungerer et al. 2010, Wilhelm et al. 2009a), their interconnects and memory controllers (see Akesson et al. 2007, Paolieri et al. 2009b, Reineke et al. 2011, Wilhelm et al. 2009b,c), and timing analyses and support for predictable execution of singlethreaded and multithreaded programs (cf. Chattopadhyay et al. 2012, Gebhard et al. 2011, Gerdes et al. 2012b, Gustavsson et al. 2012, Kelter et al. 2011, Lickly et al. 2008, Paolieri et al. 2009a, Pitter 2008, Puschner and Schoeberl 2008, Rochange 2011, Rochange et al. 2010, Staschulat et al. 2007, Thiele and Wilhelm 2004, Yan and Zhang 2008, Yoon et al. 2011).

Today's research is focused on how legacy (and) singlethreaded programs can use those newly available (COTS and predictable) embedded multi-core processors. The work presented in this thesis already steps ahead, proposing techniques to allow for executing timing analysable multithreaded parallel HRT programs on predictable shared-memory multi-core processors. While concurrent (distributed) embedded systems are already well introduced, e.g. in the automotive domain, fine-grained, thread-level synchronisation on shared-memory multi-core processors sparks the problem of timing analysability for multithreaded HRT programs. Also, one might assume that multithreaded programs do naturally clash with the isolation property, thus worsen the problem of certification. But, being tightly coupled and cooperating execution on a fine-grained level is the key difference between multitasked programs (multiprogrammed workload) versus multithreaded parallel programs; hence, the threads of a multithreaded parallel program share the same criticality level per se. Then, summing up, the problem with certification caused by interferences still arise, in the same way as for singlethreaded programs, between different multithreaded programs, but not between the threads of one parallel program.

Bringing parallelisation and HRT computing together might solve the problem of higher needs in performance while also keeping energy consumption low, however, new problems that need to be solved arise. In particular, it motivates to answer the following questions:

- How can a timing predictable parallelisation process for HRT programs targeting shared-memory multi-core processors look like?
- How can the problem of timing analysable and predictable synchronisation for tightly coupled, multithreaded programs on multi-core platforms be solved?
- Is it possible to statically analyse the timing of parallel HRT programs using thread-level synchronisation techniques producing tight and safe upper bounds?

In this thesis, answers and approaches that aim to solve the above questions are presented. The thesis on hand proposes solutions by hardware-software co-design of synchronisation techniques that can be analysed with a static timing analysis tool, revealing techniques to allow for worst-case efficient execution of parallel HRT programs. The solutions proposed for HRT capable synchronisation in hardware and software are applicable (with only small changes) in a wide range of possible future shared-memory multi-core processors. By integrating the proposed synchronisation techniques in an augmented memory controller, portability to other multi-core platforms seems highly possible, as far as some restrictions for predictability of the microarchitecture are retained. Furthermore, the presented preliminary parallelisation process embraces programmers and timing analysers allowing for transferring needed information between them, and, in the long run, targeting a complete software engineering process for efficiently parallelising HRT programs in a structured and well-defined way to enable timing predictability on multi-core processors.

1.1. Outline

The thesis on hand provides the following contributions:

A. A timing analysable shared-memory multi-core processor has been extended to allow for executing timing analysable parallel HRT programs. To achieve analysability and worst-case performance, the memory controller has been augmented with the needed logic to support atomic read-modify-write (RMW) operations in a HRT environment. An existing system software has been extended with various timing analysable software synchronisation techniques for timing predictable synchronisations in multithreaded parallel HRT programs.

B. Static timing analyses of memory latencies (worst-case memory latencys (WCMLs)), the implemented hardware/software synchronisation techniques, and parallel programs benchmarking these synchronisation techniques have been carried out on an embedded shared-memory multi-core processor.

C. An optimisation technique has been implemented in hardware to reduce the pessimism in static timing analysis of parallel programs: the split-phase synchronisation technique. It is shown that the split-phase synchronisation technique can improve the WCET guarantees of parallel programs by reducing the WCMLs of frequent loads and stores, while sacrificing latency on infrequent RMW operations.

D. First ideas and structures towards a parallelisation approach of HRT programs on multi-core processors are shortly introduced. The approach uses *parallel design patterns* and *synchronisation idioms* to increase information flow between programmers of parallel HRT programs and static timing analysers. This should help to achieve better worst-case predictability, ease the process of static timing analysis of parallel programs because of a limited design space for program structures, and the programmer is highly supported in writing timing analysable parallel programs.

This thesis is structured as follows: Chapter 2 discloses the basics on embedded real-time systems and timing analysis, and fundamentals on synchronisation techniques in hardware and software supporting the execution of parallel (real-time) programs on (embedded) shared-memory multi-core processors.

Details on the implemented HRT capable synchronisation techniques are presented in Chapter 3: it provides discussions on consistency and atomicity of RMW operations integrated in the proposed augmented memory controller, and also highlights the timing analysable software synchronisations build on top of the underlying hardware techniques. The chosen software synchronisation techniques include busy-waiting (spinning) and blocking (suspending) lock-based approaches to secure critical sections, as well as barrier implementations for progress coordination. It closes with a detailed overview on related synchronisation techniques, from the high-performance as well as embedded domain.

Besides the static computation of worst-case memory latencies on a predictable shared-memory multi-core processor (enhanced with the augmented memory controller), Chapter 4 also provides detailed static WCET analyses of the implemented synchronisation techniques. Furthermore, Chapter 4 includes static timing analyses and comparisons of parallel HRT programs applying those HRT capable synchronisation techniques; it closes with related work on timing analysis for access to shared resources, and parallel execution of real-time programs.

In Chapter 5 an optimisation of the augmented memory controller to speed up the worst-case latencies of frequent normal loads and stores over infrequent RMW operations is presented: the split-phase synchronisation technique. After introducing the basic idea of the split-phase synchronisation technique, the implementation in the augmented memory controller is described in detail. Chapter 5 finishes by detailing the impact of the split-phase technique on the worst-case memory latencies, and the WCET guarantees of parallel HRT programs.

Chapter 6 depicts an introduction on parallel design patterns and synchronisation idioms as baseline for programmers to support them in producing timing predictable parallel programs, and foster static timing analyses by providing annotations. By that, the transfer of knowledge from programmer to timing analyser should be eased, and the error prone effort of manual annotations by the timing analyser is directly derived from the used parallel design patterns and synchronisation idioms by the programmer. Thus, the information loss in analysed binary code should be highly compensated.

Finally, the thesis is concluded with Chapter 7, providing a summary of this thesis, concluding remarks, and links for future work and research.

In the appendices the source codes (Appendix A) and binaries (Appendix B) of the implemented software synchronisations are presented. Also, the control flow graphs (CFGs) of the implemented synchronisation techniques derived from the static WCET tool OTAWA, as baseline for the static timing analyses in Chapter 4, are depicted in Appendix C.

You may delay, but time will not. – Benjamin Franklin

2 Basics

This chapter covers the concepts of real-time and timing analyses in Section 2.1, and the basics of synchronisation of parallel programs, with hardware and software techniques, in Section 2.2.

2.1. Real-Time

In real-time computing systems, the system's correctness does not only depend on the computational result, but also on the correctness in time. Stankovic and Ramamritham (1990) categorise real-time systems in dimensions; one of them being the strictness of the deadlines. A deadline of a real-time task is defined as the time after which the computation needs to be finished. Depending on how critical it is to miss a deadline, real-time tasks can be separated in soft real-time (SRT) tasks, and hard real-time (HRT) tasks. The missing of some deadlines of a (periodical) SRT task, e.g. audio/video streaming, can be tolerated. Whereas for HRT tasks, the missing of a deadline leads to a situation that could harm people or cause severe damage. In the following, the distinction HRT task, non-real-time (NRT) task, and non-hard real-time (NHRT) task for tasks will be used.

A typical misconception of real-time systems is that real-time means *fast* (see Buttazzo (2004), Stankovic (1988) for a detailed collection of typical misconceptions in real-time systems). But, task deadlines (see Table 2.1) may vary between *ms*, *seconds*, or even *minutes*. Hence, it is not important how fast a HRT task executes, it is more important that a HRT task:

- finishes before its deadline under any possible circumstances, even the worst, and
- that it is timing analysable.

Thus, timing analysability respectively timing predictability (see Stankovic and Ramamritham 1990, Wilhelm et al. 2008) is the major feature to maintain in HRT systems.

TABLE 2.1.:
Typical durations of real-time tasks (flight-by-wire) in the Airbus A-340
(see Brause 2004, p. 35).

Task	Duration
Acceleration (x,y,z)	5 ms
Rotation angle	40 ms
Display refresh	1 s
Temperature	1 s
GPS position	10 s

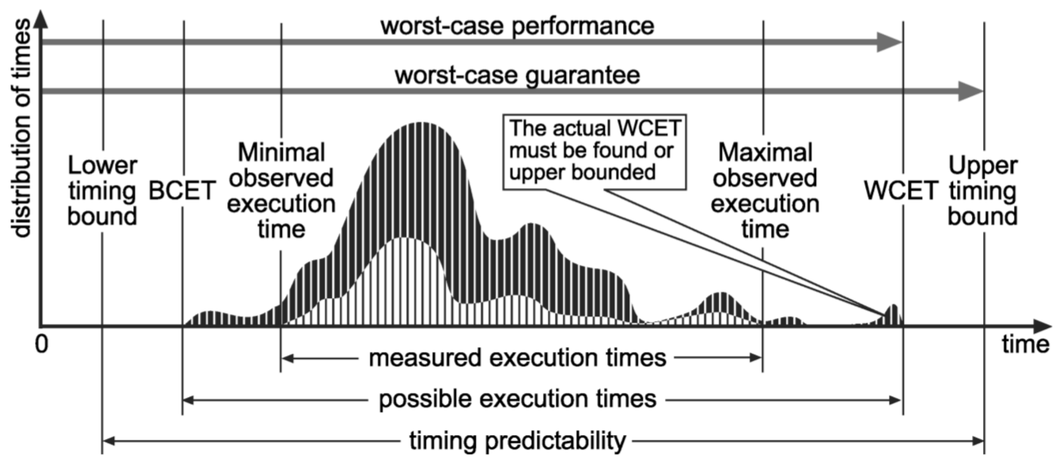


FIGURE 2.1.: Distribution of execution times and their classification (figure taken from Wilhelm et al. 2008).

2.1.1. WCET Analysis

Figure 2.1 shows an overview of different timing aspects for a real-time task. The lines show the distribution of execution times varying for different data and code coverage of the program's run. The best-case execution time (BCET) is the lowest possible timing limit, whereas the worst-case execution time (WCET) is the highest possible timing limit. Both, the BCET and WCET, are not known a-priori and very hard to determine. *Timing predictability* of a HRT computing system is defined by Wilhelm et al. (2008) by the deviation of the estimated upper and lower timing bounds. In this thesis, this is defined as *system predictability* whereas timing predictability is defined as the difference between the real WCET and the estimated WCET guarantee (in Section 4.1 a discussion on timing predictability is detailed). However, for HRT systems it is mostly more important to compute an upper timing bound, the *WCET guarantee*, of a HRT task, instead of a lower timing bound, to predict if deadline misses are possible. Also, the *WCET guarantee* is the main factor for dimensioning the needed size and performance of a HRT computing system. The maximum observed execution time (MOET) depicts the highest execution time observed for a number of runs of the program. However, it is possible that the MOET is lower than the actual WCET, as not every architectural detail might reflect in the program's runs.

Static WCET Tools

Static WCET analysis tools compute an upper timing bound, which is, in the best case, close to the real WCET. The term describing how close the computed upper bound is to the unknown WCET can be defined as *WCET tightness* or timing predictability. The computed upper timing bound is called **safe**, if $T_{Upper\ timing\ bound} \geq T_{WCET}$. Static WCET tools compute an upper timing bound using a WCET model of the whole system. Thus, the model needs to be correct and must reflect the details of the architecture.

Also, the *flow facts*, which define the control flow of a given program, need to be taken into account to determine the worst-case path. Then, based on the worst-case path, an upper timing bound is computed with mathematical methods, for instance by solving an Integer Linear Programming (ILP) problem.

In this thesis, the static WCET tool *Open Toolbox for Adaptive WCET Analysis* (OTAWA) (see Ballabriga et al. 2010) has been used for the computation of upper timing bounds, the so-called WCET guarantee. More detail on the static timing analysis with OTAWA, especially for parallel programs, is introduced in Section 4.1.3.

Other static WCET analysis tools, as mentioned by Wilhelm et al. (2008), are *aiT* (AbsInt, Saarbrücken), *Bound-T* (Tidorum, Helsinki), *Chronos* (National University of Singapore), *Heptane* (IRISA, Rennes) and various prototypes from TU Vienna, Florida State University, and Chalmers University of Technology.

Measurements and Measurement-based WCET Tools

A further set of tools to compute an upper timing bound are measurement-based WCET tools. In contrast to static WCET tools, measurement-based WCET tools are based on the measured execution time of basic blocks on the actual hardware platform. The MOETs of basic blocks are then used to compute an upper timing bound, for instance by assuming that the execution time of a specific basic block does need the maximum time that has been observed for that basic block. That is the second step is similar to the computation of static timing analysis tools, however, the resulting WCET estimates rely on the measured (observed) execution times. Therefore, measurement-based WCET tools need a high code and data coverage enabling them to compute a safe upper timing bound. However, as the computation is done on the actual hardware platform, no model of the architecture is needed.

An example of a measurement-based WCET tool, which is e.g. used for timing analyses in the industry, is the commercial tool *RapiTime* (see Rapita Systems Ltd. 2011).

2.2. Synchronisation Techniques

This section presents basics on synchronisation techniques, mostly independent of the special demands in real-time systems. Details of the HRT capable implementation of selected hardware/software synchronisation techniques is shown in Chapter 3.

In concurrent respectively parallel systems, threads compete on shared resources, e.g. the global memory in shared-memory systems, but also cooperate to gain speedups related to sequential computation of a problem. Competing threads can use mutual exclusion (see Section 2.2.1) to arbitrate their gain of access on shared resources. For cooperating threads, event synchronisation mechanisms (see Section 2.2.2) can be used (see Culler et al. 1997, Taubenfeld 2008, Ungerer 1997). However, the data in shared memories must be *consistent* to enforce functional correctness of parallel programs.

Atomicity

The term *atomicity*—also referred to as *linearisability* by Herlihy and Wing (1990)—defines that each *atomic* operation is seen, e.g. by each core in a multi-core processor, as if the operation is effective instantaneously (also see Ungerer 1997). *Atomic operations* either complete successfully or, when failing, do not have any noticeable effect. Examples of atomic operations in multiprocessors are the well-known read-modify-write (RMW) operations (see Kruskal et al. 1988, Mellor-Crummey and Scott 1991a).

Consistency

Consistency is a mandatory requirement for functional correctness in distributed computing, and also for multithreaded parallel programs. Consistency models are describing characteristics concerning for instance the order of memory operations in shared-memory systems. Hennessy and Patterson (2003) describe *sequential consistency*, introduced by Lamport (1979), as the most forthright model for memory consistency. It requires that the results of executions from concurrent running threads is the same as if the thread execute sequentially in order. Dubois et al. (1986) later introduced the notion of *weak ordering* (see Adve and Hill (1990) for a refined definition of weak ordering).
The idea of weakly ordered systems is that they appear sequential consistent by ordering accesses dispatched from different processors with explicit synchronisation operations that can be recognised by hardware. Ungerer (1997) gives a detailed overview on consistency models for multiprocessor systems, which also hold for multi-core processors.

2.2.1. Mutual Exclusion

Mutual exclusion guarantees the exclusive access of one of competing threads on a shared resource. It was originally introduced by Dijkstra (1965) demonstrating the possibility of reaching mutual exclusion with just read and write operations. In parallel programs, the following four code sections can be identified (see Taubenfeld 2008):

remainder code
: The code executed before/after the critical section, which is not part of the mutual exclusion, is labelled *remainder code*.

entry code
: The *entry code* ensures mutual exclusion in the critical section; it can also be identified as *acquire*-method.

critical section
: The *critical section* is the code section that is only allowed to be executed by one thread at a time, e.g. the sequential part of a parallel program that needs to be executed in isolation.

exit code
: The *exit code* is executed when leaving the critical section; it can also be identified as *release*-method.

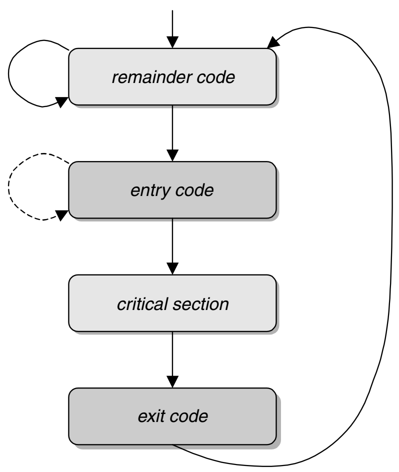


FIGURE 2.2.: Code sections for mutual exclusion in parallel programs. (image from Taubenfeld 2006).

Figure 2.2 shows a graphical representation of the above code sections (see Taubenfeld 2006). It also illustrates the whole life-cycle of a parallel program highlighting the possible contention on the entry code (dotted line), and the program flow after and before the exit code, the so-called remainder code.

Culler et al. (1997) explain mutual exclusion with an example of a room that can only be occupied by one person at a time. If a person enters the empty room, the door needs to be locked to prevent others entering it, which is synonymous with a thread successfully acquiring a lock for a critical section. All other persons trying to enter the room, will find the door closed, and it will only open after the person currently occupying it, leaves the room and unlocks the door—in other words, the thread releases the lock.

In Section 2.2.3, and Section 2.2.4 different software and hardware techniques to achieve mutual exclusion are shown. However, the *releasing/acquiring*-method of the entry respectively exit code highly depends on the used synchronisation techniques, that is the waiting algorithms. The waiting algorithms implement *blocking* or *busy-waiting* techniques (see Section 2.2.3, Table 2.2), which influence the timing behaviour of a mutual exclusion lock. Finally, the following minimum requirements on general solutions for critical sections have been identified by Dijkstra (1965):

Mutual exclusion
Only one thread is allowed to execute its critical section at a time. This needs to be assured by the *acquire*-method (entry code).

Progress
It is not allowed that blocking of other threads occurs, if one thread "is stopped well outside its critical section".

Bounded decision time
It must be assured that if more than one thread is "about to enter its critical section", the decision which thread acquires the lock is not "postponed until eternity".

Execution speed

No assumptions about the "relative speeds" of threads is allowed, nor assumptions about "their speeds [being] constant in time".

However, those minimum requirements do not establish a chronological order, and do not require bounded waiting to enter (or leave) a critical section. Knuth (1966) has shown later, that Dijkstra's requirements, while preventing livelocks, do not cover starvation-freedom. But, especially for a HRT capable implementation of mutual exclusion, it is important to enforce starvation-freedom and strong fairness between competing threads, so that an upper bound on the waiting time to enter (and also to leave) a critical section can be computed. Hence, a fifth requirement is introduced, which is based on (but stronger than) the requirement of starvation-freedom (cf. Tanenbaum 2001, p. 102):

Ordering

If more than one thread compete to enter a critical section, an analysable ordering (e.g. *first-come, first-served* (FCFS)) that allows each thread to enter (and leave) the critical section with bounded waiting must be assured.

Not all (typical) solutions for securing critical sections, even in the absence of failures and interrupts, fulfil the strong fairness, respectively FCFS-order requirement. But, without strong fairness between competing threads, the waiting algorithms are not timing predictable, thus hindering a tight WCET analysis, or even rendering it impossible. In this thesis, strong fairness between competing threads is assured by enforcing a FCFS-order with *first in, first out* (FIFO) queues. This is done in hardware (see memory bus arbitration in Section 3.1), and also in the *release/acquire*-method (entry/exit code) of the implemented software synchronisation techniques (see Section 3.3).

2.2.2. Event Synchronisation

Event synchronisations, in contrast to mutual exclusion synchronisation, describe mechanisms for cooperating threads. These mechanisms can be separated in two categories of event synchronisation (see Culler et al. 1997):

Point-to-Point

Point-to-point event synchronisation is used when two (or more) threads explicitly cooperate. An example in Bull and Ball (2003) depicts the situation of a shared array used by a number of threads with each thread holding an element of the shared array. A thread that wants to gain access to a specific element of the shared array could use a point-to-point synchronisation by only notifying the thread holding the element. Another example are so-called conditionals (see POSIX 2008), which can be used for point-to-point synchronisation, but also for global event synchronisation using broadcast.

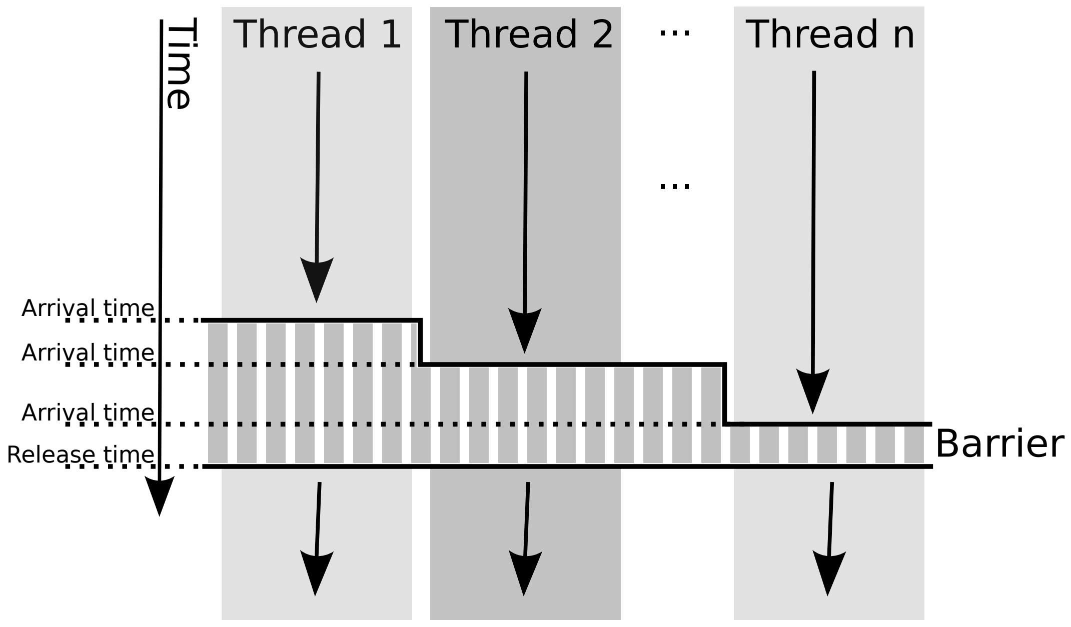


FIGURE 2.3.: Global event synchronisation: an example of n threads synchronised at a barrier with corresponding arrival times and release time of all n threads when the barrier condition is fulfilled.

Global

The *global* event synchronisations involve all or a specific group of cooperating threads. Also categorised in *group* and *barrier* event synchronisation. A typical example is a synchronised start or re-start of threads at a barrier (see Figure 2.3).

Event synchronisation is often used for parallel programs structured with producer-consumer chains, or using barrier synchronisation (see Section 2.2.3) for delineation between different sections in the parallel program (see Figure 2.3).

2.2.3. Software Synchronisation Techniques

Software synchronisation techniques are the programmer's way to use synchronisation in a simple way. The system software or Real-Time Operating System (RTOS) provides the interfaces for those synchronisation techniques for the programmer. Mellor-Crummey and Scott (1991a) present a detailed overview of common software synchronisation techniques (see also Hennessy and Patterson 2003, Ungerer 1997), and also standards like POSIX (2008) provide skeletons for commonly used synchronisations. However, Chapter 3 depicts the details on a HRT capable implementation of the below introduced software synchronisation techniques.

Software synchronisation techniques are distinguishable as either busy-waiting or blocking (also called suspending). Busy-waiting, also called spinning, is a technique to repeatedly check for a specific condition.

For instance, it is used to spin on a value in the local or global memory until a specific value is retrieved. Therefore, if a busy-waiting technique is used for mutual exclusion, the entry code might be a loop that continues until the lock is acquired. A typical example is the commonly used spin lock implemented with the test-and-set (TAS) primitive (see Section 3.3.1). Contrarily, blocking synchronisation techniques use a waiting algorithm as acquire-method. Thus, this requires some sort of scheduling for the waiting algorithm (see Sections 3.2.3, 3.3.3, 3.3.5). The advantage of blocking techniques is that it is possible to spare computational resources by e.g. suspending tasks, instead of having them spin and constantly access the shared global (off-chip) memory. However, blocking synchronisation techniques require more complex logic and scheduling, and might sacrifice performance on short critical section. In some environments or for kernel operations inside an Operating System (OS) it might not be possible at all to use blocking synchronisation techniques.

Table 2.2 shows the categorisation of commonly used software synchronisation techniques as either busy-waiting or blocking. Spin locks and ticket locks, which are typically used for short critical sections, are commonly implemented as busy-waiting, whereas mutex locks, semaphores and barriers are commonly implemented in a blocking manner.

Spin locks are mainly meant to be used internally in a system software or RTOS, and not by a programmer. This is, as they are implemented in a busy-waiting manner, and a programmer would need great knowledge of the underlying hardware primitive and architecture to use busy-waiting synchronisations in an optimal manner. In contrast, blocking synchronisation techniques, namely mutex locks, semaphores, and barriers, are the main software synchronisation techniques which ought to be used by a programmer. Nonetheless, in embedded systems, programmers often employ low-level, close to hardware languages and also have deepened knowledge of the underlying architecture and architectural features. So, the use of busy-waiting synchronisation techniques, or even inline assembling is not recognised as unusual (see also the WCET analysis results presented in Section 4.3.4).

Table 2.2.:
Comparison of commonly used software synchronisation techniques and the typical use of either busy-waiting (spinning) or blocking (suspending) methods to coordinate the entry of a critical section or respectively for event synchronisation. The synchronisation techniques marked with (x) could also be implemented using a busy-waiting strategy.

Synchronisation technique	Busy-waiting	Blocking
spin lock	x	-
ticket lock	x	-
mutex lock	(x)	x
semaphore	(x)	x
conditionals	(x)	x
software barrier	(x)	x

The summary list below depicts the most common software synchronisation techniques for mutual exclusion and event synchronisation.

Spin locks
Spin locks are easy and commonly used busy-waiting locks, which spin on a memory location, e.g. local or off-chip memory, until a specific condition is fulfilled, e.g. a specific value has been read from memory. Spin locks are mostly used internally by the RTOS to secure only very short critical sections, and to build more complex locks, namely mutex locks. They are mostly not intended to be used explicitly by the programmer because of the lack of fairness (see also Section 3.3.1).

Queued locks
Queued locks, as e.g. from Anderson (1990), MCS locks (see Mellor-Crummey and Scott 1991a) or CLH locks (see Craig 1993, Scott and Scherer 2001), have been introduced to overcome the lack of fairness in spin locks. They are either based on cache coherence protocols, or complex interactions with other core's local memories.

Ticket locks
Other locks that have been introduced to overcome the problems of spin locks are Mellor-Crummey and Scott's ticket locks. They are based on Lamport's bakery algorithm, and are used in the Linux Kernel since version 2.6.25 (January 2008). Details on their implementation are presented in Section 3.3.4.

Mutex locks
Mutex locks are a blocking synchronisation technique commonly used to enforce mutual exclusion. The waiting algorithm of a mutex lock implementation needs to take care of managing waiting threads in an ordered list to achieve fairness and progress. Details on a HRT capable implementation are presented in Section 3.3.3.

Semaphores
Semaphores were invented by Dijkstra (1968) for controlling multiple accesses to shared resources. Usually, semaphores are categorised as either counting semaphores or binary semaphores. Counting semaphores allow an arbitrary value as counter, e.g. for a number of resources that are free to be commonly used. Contrarily, binary semaphores restrict the value to be either '0' or '1', and are semantically similar to mutex locks (see Section 3.3.5), despite they do not employ the concept of a lock owner as mutex locks should according to POSIX (2008).

Software barriers
Software barriers are a useful construct for event synchronisation, namely to synchronise starting or re-starting of threads at a specific point, e.g. to organise progress in different phases of a parallel program (see Hennessy and Patterson 2003, Ungerer 1997). They are more flexible as hardware barriers, and employ lock techniques or RMW operations like fetch-and-increment (F&I) (see Section 3.3.6).

2.2.4. Read-Modify-Write Operations

Modern instruction set architectures (ISAs) provide instructions performing reads and subsequent writes atomically. These instructions are called RMW operations, where modifying stands for logic that is executed between the read and the write. This logic might be an arithmetic calculation, as needed for fetch-and-add (F&A) operations, or a comparison as in compare-and-swap (CAS) operations. Other atomic operations, like TAS or swap, are used for an atomic read and subsequent write access. A more detailed overview on RMW operations can be found in articles of Kruskal et al. (1988), Mellor-Crummey and Scott (1991a) or in *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson (2003).

In the following, the basics of the main RMW operations are given.

Test-and-Set

TAS reads a value from a memory address and writes subsequently a fixed value, e.g. a ’0’ or ’1’, to the same memory address. No other write access to the same memory address is allowed between the read and the write to maintain atomicity.

◦ **Test-and-Test-and-Set**
A variation of TAS is a primitive with an additional test operation: the test-and-test-and-set primitive (Rudolph and Segall 1984). The additional test operation is used to reduce the contention which might occur when spinning on a shared resource, e.g. the off-chip memory. The additional test operations can then be used to spin on a local memory value, and the TAS will be issued after the first test operation succeeded.

Fetch-and-Add

F&A returns the read value and modifies it by adding a constant, that is either a positive or negative value, and subsequently writes back the sum of constant and read value. It was first introduced by Gottlieb et al. (1983) as *Replace-Add* and later renamed by Gottlieb and Kruskal (1981) to the more descriptive term Fetch&Add.

◦ **Fetch-and-Increment** and **Fetch-and-Decrement**
Variations of the F&A operation are the fetch-and-increment/fetch-and-decrement (F&I/F&D) primitives. Instead of allowing to add an arbitrary constant to the read value, only adding a ’1’—called F&I—respectively subtracting a ’1’—called fetch-and-decrement (F&D)— is allowed.

Swap

The `swap` operation atomically swaps a data value from the register set with a value from the memory, in most cases the off-chip memory. Hennessy and Patterson (2003) also use the name *atomic exchange* for the `swap` operation.

Compare-and-Swap

The CAS primitive uses three parameters: the new value, the old value, and the memory address. It atomically updates the current value at the given memory address, if, and only if, the current value matches the old value parameter (see e.g. IBM z/Architecture 2005). The need for three operands, and therefore a minimum of three registers for the three parameters of CAS, could be considered a drawback of this operation. CAS is also known as *compare-and-exchange* in the x86/Itanium ISA. A variation is the multi-word compare-and-swap (CASN) operation (also called MCAS) used to update a set of matching values (see Ha-Hoai and Tsigas 2003, Harris et al. 2002).

Load-Linked/Store-Conditional

The load-linked/store-conditional (LL/SC) primitive, introduced by Jensen et al. (1987), consists of two separated instructions: a load operations (load-linked) and a store operation (store-conditional) that is guaranteed to fail if the value loaded by a load-linked operation is accessed before the associated store-conditional operation. In contrast to a CAS, it also fails when the value has been restored. Therefore, a LL/SC is regarded stronger than a CAS operation. However, most implementations in modern ISAs limit other memory accesses between a load-linked operation and the corresponding store-conditional operation, so the implementations are called *weak* or *restricted* LL/SC (see Michael 2004). Paap and Silha (1993) state that the LL/SC primitive can be implemented to emulate a CAS primitive in uniprocessor systems. As stated by Herlihy and Moss (1993), LL/SC can be seen as baseline for the concepts of *transactional memory* (see also Section 3.3.7).

Hardware Barriers

A possible implementation of hardware barriers by using a single wired-AND line is described by Culler et al. (1997). But, the authors also highlight that hardware barriers are problematic when dynamically changing the number of processes participating in a barrier. Thus, hardware barriers are very inflexible for the use in parallel programs, and hence bus-based multiprocessors usually do not feature hardware barriers (see also Section 3.3.6).

Table 2.3 shows different Reduced Instruction Set Computing (RISC) ISAs and the RMW operations they support. The ARM, MIPS, and PowerPC ISAs all support a LL/SC operation, but neither of them implements the theoretical *strong semantic*. In the *strong semantic*, a LL/SC is required to fail the conditional-store not only on every concurrent access to that memory location, but for instance also on context switches. Therefore, their LL/SC implementations are considered as *restricted* LL/SC (additional handling of context switches and memory page faults for LL/SC is needed). Also, for a correct implementation of LL/SC, load-linked operations need to be monitored at memory level (local and global memory) so that it can be decided if a conditional-store is successful or not.

One major drawback of LL/SC operations is that the store-conditional might fail on concurrent accesses to a loaded value in a multi-core processor. Further details on the problem of bounding the execution and waiting time of the LL/SC primitive is presented in Chapter 3 in Section 3.2.4. Michael (2004) and Gao and Hesselink (2007) show that e.g. for lock-free algorithms, CAS would be sufficient over *restricted* LL/SC implementations. The split-phase synchronisation introduced in Chapter 5 is a similar technique as LL/SC, splitting the load and store phases, however, its target is to reduce the WCET pessimism in static WCET analyses.

Table 2.3.:
Supported RMW operations of different RISC ISAs in the embedded domain.

ISA	TAS	F&A	F&I/F&D	swap	CAS	LL/SC
ARM v7(ARMv7-M ISA 2010)	-	-	-	x[1]	-	x
MIPS32 v3.0(MIPS32 ISA 2003)	-	-	-	-	-	x
Power v2.06(PowerPC ISA 2010)	-	-	-	-	-	x
SPARC v9(SPARCv9 ISA 1994)	x	-	-	x	x	-
TriCore v3.1(TriCore ISA 2008)	x	-	-	x	-	-
SuperH-2A(SuperH-2A ISA 2010)	x[2]	-	-	-	-	-
MERASA(cf. TriCore ISA 2008)	x[3]	x[3]	x[3]	x[3]	-	-

The TriCore (single-core processor) ISA supports a *load-modify-store* (LDMST) operation, that is an atomic operation allowing for use a bitmask to modify a value. It also includes a *store bit* instruction (ST.T) that atomically changes one bit of a loaded word.

The MERASA[4] multi-core processor (cf. Paolieri et al. 2013, Ungerer et al. 2010) implements the TriCore ISA and reuses the `swap` primitive to implement different RMW operations in the augmented memory controller. The primitives (TAS, F&A, and F&I/F&D) are then encoded in the augmented memory controller, while the original semantic of a `swap` operation is not retained. Hence, a usual swap, as implemented in the TriCore ISA, cannot be performed by the `swap` instruction in the MERASA processor after the modification. Details on the implementation of RMW primitives in the MERASA processor, the augmented memory controller, and specifics of related work and the support of synchronisation operations in commercial off-the-shelf (COTS) multi-core processors are presented in detail in Chapter 3. In Section 3.2.4 the possibilities of implementing and reusing instructions of further ISAs, namely for ARM and PowerPC ISAs, are discussed.

[1]Only the ARMV7-R ISA and some older ARM ISAs support a *swap* operation.

[2]Only pseudo-TAS; the read instruction on a semaphore register automatically triggers a write of a '0' in hardware (also see discussion below)

[3]The implemented primitives are all based on the TriCore *swap* primitive and are encoded to other primitives in the augmented memory controller (see below and in more detail in Section 3.2.1).

[4]The *Multi-core Execution of Hard Real-time Applications Supporting Analysability* (MERASA) project has been funded by the European Union as FP-7 project under grant agreement no. 216415. See also the project website at http://www.merasa.org

In some architectures, shared atomic registers are used, instead of RMW operation, to secure critical sections, respectively to implement software synchronisations. In the SuperH-2A ISA (2010), respectively the dual-core processor SH7265 from Renesas Electronics[5], the instruction *TAS.B* is used to atomically access special function registers called *semaphore registers*. The semaphore registers should be initialised with the value '1' (signalling a free resource), and the hardware automatically writes a '0' after reading the semaphore register value. Therefore, the semaphore registers can be used to implement a TAS-like primitive. Also, it is allowed to write values to the semaphore register, e.g. to write a '1' into a semaphore register to signal that the resource is free again. The Intel Single-chip Cloud Computer (SCC)[6] (see Mattson et al. 2010) also uses this kind of registers—called TAS registers–for synchronisation shared between two cores in each tile. Also, the V850E2/MN4 (2013) ISA offers support for mutual exclusion with shared registers. In other architectures, e.g. in the Cray X-MP processor[7], *semaphore registers* shared between processors are used for interprocess communication (see for details Ungerer 1997, p. 258f.).

But, these approaches have several drawbacks for being used in timing predictable multi-core processors. Firstly, they exhibit a hard limit for the number of concurrently possible locks, i.e. the number of available shared registers. Secondly, they only allow the usage of simple TAS locks, however, in combination with software techniques it is possible to implement more complex synchronisations. Though, this reduces the efficiency, especially in the worst-case, due to pessimism in the WCET analysis from possible concurrent accesses to the shared registers. Another drawback is the rising complexity for accessing shared registers from an increasing number of cores. Either, a specific arbitration logic is needed (e.g. as in the SuperH-2A ISA (2010)) so that only one access per core is allowed to the shared registers, or a crossbar solution could be used with a number of read and write ports at the shared control registers, as e.g. in the CarCore processor (cf. Mische et al. 2010, Uhrig et al. 2005). Both approaches have their disadvantage either in time for accesses (overhead from arbitration), or in space and hardware costs (crossbar/ports).

Another technique for fine-grained control of shared data in a memory stems from the supercomputer domain: full/empty bits in hardware (see for more details Smith 1981, Ungerer 1993, p. 325f.). Full/empty bits are e.g. used in the multithreaded multiprocessor HEP (see Kowalik 1985), and in the many-core processor Godson-T (see Fan et al. 2012). For synchronisation in the HEP supercomputer, the ISA includes specific atomic load and store operations. For instance, load operations are implemented to either wait to complete after a specific state of the accessed memory cell is reached (full), or fetch the data from the memory cell and change its state to empty. On the other hand, store operations can check the state before writing data to a memory cell, and only write if the state of the memory cell is empty and set it to full, otherwise the store operations waits for the state to change from full to empty.

[5]see `http://www.renesas.com/products/mpumcu/superh/sh7260/sh7265/index.jsp` [last retrieved: April 2013]

[6]see `http://communities.intel.com/community/marc` [last retrieved: April 2013]

[7]see `http://www.cray.com/About/History.aspx` [last retrieved: April 2013]

This technique is further detailed by Smith (1981) and Ungerer (1993). The Godson-T provides beside full/empty bit support in hardware also a synchronisation manager (SM) which handles lock-based synchronisation with queues in hardware to speed up parallel (scientific) programs in the high-performance domain.

Similar to the concept of full/empty bits are the I-structures (see Arvind et al. 1989) and M-structures (see Barth et al. 1991) from the dataflow computing domain (see also Ungerer 1993, p. 156f.). I-structures and M-structures are implemented in the parallel computing, dataflow language *Id* (see Nikhil 1991). Arvind et al. (1989) describe I-structures in detail. They state that two operations on an I-structure storage memory are provided: an I-fetch tries to read a value, and I-store tries to update a value. Each value in the I-structure storage is tagged with a status bit that is either 'empty' or 'non-empty', similar to full/empty bits. The I-structure storage controller checks for every access the state of the tag bit. I-fetch operations are queued by the controller, if the location is marked as 'empty', or return a value for 'non-empty' tagged locations. The tag bit is only set to 'non-empty' when a single I-store operation to an 'empty' location is recognised; then, queued I-fetch operations are provided (with no specific order) with the updated value. If an I-store operation on a 'non-empty' location is detected by the I-structure storage controller, it generates a runtime error.

M-structures are a similar construct supporting atomic updates, however, introducing imperative data structures in the functional (declarative) language *Id*; this allows, as stated by Barth et al. (1991), for more efficient implementations for specific problems. The two operations *take* and *put* provide implicit synchronisation, but require local barrier synchronisation to enforce a correct ordering of operations (if not provided by strictness of operations in *Id*). Also, Barth et al. (1991) denote that the handling of the *put* operation (that writes a value to an 'empty' location) can be extended to allow for multiple *put* operations on an 'empty' location being queued, e.g. for producer-consumer interactions. Ungerer (1993) expounds that I-structure storage controller can also support the M-structure operations *take* and *put*.

Related M-structure operations called E-fetch and E-store were introduced at the same time—independently, as purported by Barth et al. (1991)—by Milewski (1990), based on the previous work of Arvind et al. (1989). E-fetch operations are implemented similar to M-structures *take* operations; after a successful E-fetch, the data location is tagged 'empty'. The E-store operation, however, is a bit different to the I-structures I-store operation respectively the M-structures *put* operation. While Milewski (1990) also implemented E-stores to disallow the breaking of the "single assignment rule", the waiting queues with E-fetches are handled differently than for M-structures, that is only the first waiting request is served. Milewski 1990's E-fetch and E-store operations do need a similar support with a storage controller in hardware as I-structures or M-structures.

Further related primitives used for synchronisation and atomic RMW operations are the LL/SC pairs, e.g. provided by ARMv6-M ISA (2010), ARMv7-M ISA (2010), MIPS32 ISA (2003), PowerPC ISA (2010), and the CAS primitive, e.g. provided by the SPARCv9 ISA (1994); they are presented and discussed in detail in Section 3.2.4.

In Section 3.3.7 a broad view on software synchronisation techniques (also including hardware support) are presented in detail.

Procrastination is the thief of time. – Edward Young

3 Hard Real-Time Capable Synchronisation

This chapter presents how HRT capability and timing predictability is achieved by collaborating hardware and software techniques, that is timing predictable, atomic RMW operations in hardware and worst-case efficient synchronisation techniques in software build on top of those hardware implementations. In particular, in Section 3.1 the range and baseline of possible timing analysable synchronisation techniques for embedded multi-core processors are presented. Section 3.2 depicts the augmented memory controller which includes the synchronisation logic for atomic RMW operations. In Section 3.3 it is highlighted how HRT capability and timing predictability for commonly known software synchronisation techniques is achieved, like e.g. locks, semaphores, and barriers.

Static timing analyses of the proposed HRT capable synchronisation techniques are done in Chapter 4. In Chapter 4 also the computations of worst-case memory latencys (WCMLs) for the *augmented memory controller* are detailed. In Chapter 5 details on the split-phase synchronisation technique, that is an improvement for reducing the pessimism in the WCET guarantees of parallel programs, are shown. The logic of the split-phase synchronisation technique is also implemented in the augmented memory controller.

Please note that part of this chapter is based on a publication in the proceedings of *Design, Automation and Testing in Europe (DATE)* 2012 (see Gerdes et al. 2012b).

3.1. Synchronisation in Embedded Multi-Core Processors

HRT embedded multi-core processors need to fulfil different requirements than today's COTS multi-core processors aim at. As for most COTS processors high average performance is the key design goal, it is timing predictability for HRT capable processors. However, predictability in HRT capable multi-core processors comes with a price. Actually, HRT multi-core processors are either specifically engineered, as e.g. shown by Akesson et al. (2007), Liu (2012), Liu et al. (2010), Paolieri et al. (2013, 2009a), Ungerer et al. (2010), Wilhelm et al. (2009b), or specific restrictions for the access to shared resources, mostly also accompanied by a complex WCET analysis, are introduced, for example in Kelter et al. (2011), Yan and Zhang (2008) (see also Section 4.5).

To fulfil requirements for safety and security, various safety standards have been specified, and the developed software must be certified against them: the *ISO 26262* in the automotive domain, the *EN 50128* for railway systems, the IEC 61513 for nuclear power plants, the *IEC 62061* for the machinery domain (all based on the *IEC 61508* standard[1]), and the *DO-178B* in the avionic domain. Therefore, deterministic execution models on COTS multi-core processors is an emerging research field, e.g. the research of Boniol et al. (2012) and Nowotsch and Paulitsch (2012) on the Freescale P4080, as well as research on timing analysable and predictable multi-core architectures. Further COTS multi-core processors for the safety-critical domains are the ARM11 MPCore, the Renesas SH7265 (SuperH-2A ISA 2010) and V850E2/MN4 (2013), and the Infineon AURIX (2013) multi-core processor.

[1]see http://en.wikipedia.org/wiki/IEC_61508 for a rough overview on those safety standards.

However, all the above mentioned COTS multi-core processors are not designed to enable timing predictable execution of real parallel, multithreaded HRT programs. Beside being still very difficult to analyse with a static WCET tool, they provide different synchronisation techniques for loosely coupled tasks, that is for multiprogrammed workload. The switch to multithreaded parallel HRT programs is still to come, and yet the needed effort to find suitable timing analysable synchronisation techniques and their hardware and software implementations to be researched. In the following, a possible hardware technique to ease timing predictable synchronisation in multithreaded parallel HRT programs is presented, followed in the next section by presenting software synchronisation techniques that rely on the proposed hardware techniques. Together, this approach yields to allow for timing predictable execution of tightly coupled, multithreaded parallel HRT programs on shared-memory multi-core processors.

3.1.1. Hard Real-Time Capable Multi-Core Processors

An embedded multi-core processor that is HRT capable has to fulfil certain requirements. The design goal is often to isolate the HRT parts and tasks as much as possible, and use of real-time capable arbitration techniques when accessing shared resources, e.g. the shared memory, is needed. In this thesis, a WCET model of a HRT capable multi-core processor, the MERASA processor (see Paolieri et al. 2013, Ungerer et al. 2010) is used for the static timing analysis with OTAWA. The modelled multi-core processor features a configurable number of in-order, dual-issue simultaneous multithreading (SMT) cores. The use of SMT techniques enables it to execute mixed-critical programs consisting of HRT and NHRT threads. Moreover, isolation of HRT and NHRT threads in the cores preserves the HRT capability (see Mische et al. 2010).

The memory controller and interconnect cannot isolate concurrent accesses of different cores. Instead, the use of isolation here would require partitioning the global memory, as well as duplicating shared resources, and thus hinder or even disallow communication between threads. Therefore, the use of shared resources should be allowed.

Interferences are handled by an upper bounding of accesses to shared resources enforced by a real-time capable bus as interconnect to memory and cores, as well as a real-time capable memory controller (see Paolieri et al. 2009a). Scratchpad memories are used as local memories for each core, namely a data scratchpad (DSP) and a dynamic instruction scratchpad (D-ISP) (see Metzlaff et al. 2011), but no caches for the HRT threads. Figure 3.1 depicts a schematic overview of the MERASA multi-core processor. It is binary compatible with the Infineon TriCore architecture (see TriCore ISA 2008), which is widely used in the automotive and construction machinery domain.

In the MERASA project, a high-level processor simulator, a cycle-accurate SystemC simulator, and a Field-Programmable Gate Array (FPGA) prototype of the MERASA processor have been built. The simulators offer interfaces to use the measurement-based WCET tool RapiTime. Furthermore, a WCET model of the MERASA processor has been developed for static WCET analysis with OTAWA. An important design decision—augmenting the memory controller with logic supporting synchronisations—is presented in detail in the following, and WCET analyses in Chapters 4 and 5.

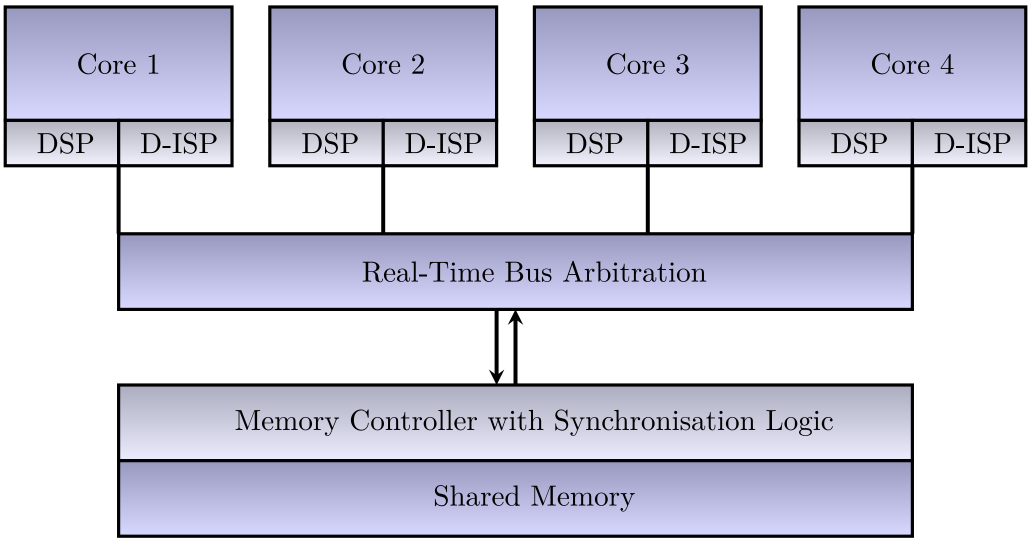

Core 1
DSP
D-ISP
Core 2
DSP
D-ISP
Core 3
DSP
D-ISP
Core 4
DSP
D-ISP
Real-Time Bus Arbitration
Memory Controller with Synchronisation Logic
Shared Memory

FIGURE 3.1.: Overview of a quad-core MERASA processor, a predictable, bus-based, shared-memory multi-core processor, stressing the embedded synchronisation logic in the memory controller.

Restrictions

For the static WCET analyses in this thesis the baseline is restricted to only execute one single HRT thread per core. A parallelised task or program consists of multiple threads, but only one HRT program is executed in concert with potential NHRT programs. Therefore, all parallel threads of one HRT program share the same, highest priority.

Also, it is assumed in this thesis that no interrupts (for HRT threads) occur. Solutions to interrupt handling in HRT systems—related to aperiodic task execution in the domain of real-time scheduling—is either to disable interrupts and only allow them in specific time slots (see Lehoczky and Ramos-Thuel 1992) or to include the overhead in the WCET of the corresponding task through utilisation bounds and off-line bandwidth preserving (see Abdelzaher et al. 2004, Liu and Layland 1973, Spuri and Buttazzo 1994). However, Sandström et al. (1998) state that this might lead to high overestimations in the estimated WCET or only works for higher response times, and propose a different solution to cope with interrupts in statically scheduled HRT systems.

The starting of threads of a parallel HRT program needs to be synchronised in a timing analysable multi-core processor, that is all threads should start at the same time on each core. In the MERASA processor this has been achieved by creating threads with a suspension bit in the thread control block (TCB). When all threads are ready to run, the suspension bit is deleted by a hardware priority manager (cf. Wolf et al. 2010b). The same behaviour is enforced at other synchronisation points, e.g. at barriers.

Furthermore, for the synchronisation techniques presented in this chapter the absence of failures is assumed. However, approaches for concurrent programming problems with fault tolerance exist, e.g. the work on non-blocking algorithms (see Section 3.3.7).

3.1.2. Hardware Support for Synchronisation Techniques

Synchronisation techniques are often implemented in abstraction layers for the programmer, e.g. in an OS, respectively an RTOS or a firmware/system software. However, for testing and debugging, or if specific control on a low-level hardware layer is needed, bare metal programming can be and is used. On many COTS multi-core processors, programming the *bare metal* is the sole solution to achieve deterministic execution and timing predictability on those platforms (see Boniol et al. 2012, d'Ausbourg et al. 2012).

Bare metal programming became famous, as it defines the start of the Linux development by Linus Torvalds (see Torvalds and Diamond 2002). With the upcoming of multi-/many-core processors, programming the bare metal becomes highly interesting again, and often it is the only way to program a new device for which no OS yet exists[2].

Consistency and atomicity are key aspects for synchronisation in multi-core processors (cf. Sections 2.2, 3.1.4 and 3.2). Therefore, these requirements must be fulfilled, e.g. with the help of atomic hardware primitives in the architectures. The underlying hardware architecture mostly allows for using different primitives for consistent and atomic synchronisation. More details on how worst-case efficient atomicity for such RMW primitives in a multi-core processor can be achieved are discussed in the next Section 3.1.4. These primitives are made available through the hardware on diverse software layers that is either bare metal, firmware/system software support, or a complete RTOS.

Bare Metal programming is mostly used for testing and debugging of "new" architectures and systems. The bare metal layer is in the most cases just above the Basic Input Output System (BIOS) or after a boot loader sequence, and allows to access the memory or the interconnect through lightweight and simple interfaces. Then, higher abstracting software, like a firmware, a system software or an OS are build from there respectively on top of it. But, bare metal programming is also used to gain specific control over how something is done, that is hardware-near programming for efficient solutions.
Synchronisations that can be used on this level in multi-cores are mostly limited to:

- Peterson's algorithm(see Peterson 1981),
- Dekker's algorithm (cf. Ungerer 1997, page 120ff.),
- shared registers for mutual exclusion, and
- RMW primitives for atomic read/write accesses.

Firmware/System Software provide an abstraction layer for programmers, but still allow for hardware-near programming. Many embedded devices are running programs which are directly compiled with the firmware/system software. Synchronisations that can be used on this level are the same as for bare metal programming, but adding also software synchronisations, e.g. over application programming interfaces (APIs) like POSIX (2008). However, the use of hardware-near techniques like shared registers or RMW primitives might be limited to encapsulate the use of synchronisations in the firmware/system software.

[2]The website `http://sites.onera.fr/scc/baremetal` presents information on bare metal programming on today's and future's many-core platforms, e.g. for the Intel SCC (see Mattson et al. 2010).

(Real-Time) Operating Systems offer an even higher abstraction level for programmers. The synchronisation methods that can be used on this level are mostly the same as above, and the use of hardware-near techniques, like shared registers or RMW primitives, to encapsulate the use of synchronisations in the RTOS might be even more restricted as for a firmware/system software.

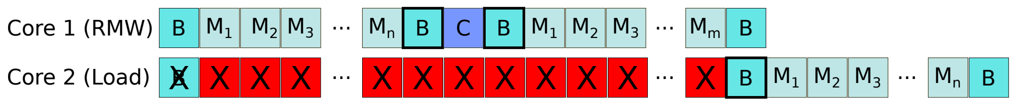
3.1.3. Consistency in the MERASA Multi-Core Processor

Sequential consistency, introduced by Lamport (1979), has two requirements: (R1) each processor issues memory requests in the order specified by its program, and (R2) memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. The HRT capable MERASA multi-core processor fulfils those two requirements through the arbitration in the cores (R1) and the augmented memory controller (R2) (see Gerdes et al. (2012a,b), Ungerer et al. (2010)). Furthermore, bringing the requirements for weakly ordered memory operations stated by Dubois et al. (1986) together with the MERASA multi-core processor: 1) accesses to global synchronisation variables are strongly ordered, 2) no access to a synchronisation variable is issued in a core before all previous global data accesses have been performed and 3) no access to global data is issued by a core before a previous access to a synchronisation variable has been performed. One of the major issues for consistency using synchronisations under a weak consistency model is then atomicity of RMW operations (detailed in the following).

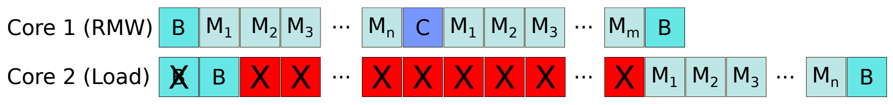
3.1.4. Atomicity of RMW Operations

The use of synchronisation techniques (see the work of Kruskal et al. (1988) for a survey on software synchronisations, and Molesky et al. (1990) for implementations of predictable synchronisation techniques in multiprocessors respectively in multi-core processors by Gerdes et al. (2012b)), for instance to avoid data races, is mandatory for functional correctness of parallel programs with locks.

Atomicity can be achieved if the access to the hardware structure or memory region that provides synchronisation variables assures mutual exclusion, e.g. shared atomic registers that can be only accesses by one thread at a time. Despite the arguments discussed already in Section 2.2.4, such shared registers only offer binary values, which might be sufficient for access control to peripheral devices in concurrent systems, but do not offer more complex synchronisation support needed for multithreaded parallel HRT programs on shared-memory multi-core processors. Thus, a more general solution is needed to implement worst-case efficient synchronisation for a shared-memory multi-core processor. One common solution is to use software synchronisation techniques in parallel HRT programs with the support of hardware-implemented RMW operations. *Atomicity* ensures that an operation consisting of a read, a modification and a write cannot be interrupted, and is executed completely. For a bus-based, shared-memory multi-core two different possibilities to implement atomicity for RMW operations are conceivable: 1) locking the interconnect and modify in cores, or 2) logic for atomic operations outside of the cores in a shared resource.



(a) Locked interconnect



(b) Augmented memory controller

FIGURE 3.2.: Memory access pattern for RMW operations with a locked interconnect (a) respectively with the augmented memory controller (b).

The latter one could be either implemented in the memory controller of the shared memory, which is further detailed in Section 3.2 (similar as in older multiprocessor architectures, e.g. in the NYU Ultracomputer presented by Almasi and Gottlieb 1989, p. 435), or as a dedicated shared memory for synchronisations, e.g. at the interconnect like shared L2 caches in high-performance systems (see also related work in Section 5.2.5).

Figure 3.2 depicts the memory access pattern of those two solutions and the interference that is caused from RMW operations on other memory operations of concurrently running HRT threads. In detail, Figure 3.2 shows the memory access pattern cycle by cycle, that is the bus access (labelled 'B'), the actual memory accesses (M_1 to M_n represent a load operation, and M_1 to M_m a store access), and the modification of the read value (labelled 'C'). The red squares (labelled with 'X') depict the situation in which the memory accesses of the concurrent executed thread of another core (Core 2) is blocked from the memory accesses of the core issuing a RMW operation (Core 1).

With a locked interconnect, all request of concurrent threads to the shared memory need to stall at the core-level until the RMW operation issued and completed the write operation. The bold squares in Figure 3.2(a) show the additional bus accesses that are needed with a locked bus compared to the augmented memory controller (see Figure 3.2(b)). Hence, the worst-case latency for a normal load or store with a locked interconnect increases, while it is lower with the embedded logic in the memory controller. Thus, embedding the RMW logic in the memory controller helps to achieve atomicity without too much latency sacrificed from the impact of slow synchronisation accesses, that is reducing the WCMLs accessing the shared memory (see detailed static timing analyses in Chapter 4). Also, the computation phase with a locked interconnect (blue 'C' square in Figure 3.2(a)) might be longer than the depicted one cycle, depending on how the modification of the RMW operation is done. Additionally, Wolf et al. (2010a, 2011) have shown that locking the memory bus to prevent other cores interrupting a RMW operation also leads to high contention for all cores and threads in accessing the shared resources, namely the shared memory or Input/Output (I/O) devices, resulting in high pessimism for the WCET guarantees.

Further discussions on atomicity and consistency of RMW operations concerning the split-phase synchronisation technique are detailed in Section 5.1.1.

3.2. Augmented Memory Controller

The aim of the augmented memory controller, including the needed logic to execute RMW operation, is two-folded. Firstly, the approach eases the use of today's available or future ISAs by reusing already implemented instructions, so, the generality of the approach is retained. Secondly, it assures worst-case efficiency while maintaining functional correctness of accesses to shared resources (see Sections 3.1.3 and 3.1.4).

3.2.1. Implementation of the Augmented Memory Controller

The augmented memory controller (see Figures 3.1 and 3.3) implements the needed logic of atomic RMW operations for synchronisations. In this thesis, the detailed physical implementation of a predictable Synchronous Dynamic Random-Access Memory (SDRAM) controller is out of scope, but an existing SDRAM controller should mostly not be affected by the proposed augmentation with synchronisation logic. Figure 3.3 depicts the separation of added synchronisation logic and SDRAM controller, and the interconnect to the real-time bus and the SDRAM memory. Akesson et al. (2007), Mutlu and Moscibroda (2007), Paolieri et al. (2009b), Reineke et al. (2011), Whitham and Audsley (2009), and Lakis and Schoeberl (2013) propose different solutions concerning predictable Dynamic Random-Access Memory (DRAM) access and the (detailed physical) implementation of predictable SDRAM controllers. These could then be interfaced with the augmented part in the memory controller as shown in Figure 3.3. The augmented memory controller has been exemplarily implemented in the MERASA SystemC simulator[3] with a simplified SDRAM controller for proof of concept. Also, some proposed techniques are implemented in the MERASA FPGA prototype[3], for instance to derive realistic memory latencies.

In the following, the reusing of instructions in the TriCore-based MERASA ISA is described, but it is possible to apply the presented techniques to other ISAs as well. The `swap` instruction of the TriCore ISA (2008) is usually used (in the single-core TriCore) to atomically swap a data register value with a memory word. As the data, which is usually swapped by the `swap` instruction, is not needed for the implemented RMW operations it is used to encode the corresponding RMW operations. In the augmented memory controller the data value can be decoded to identify a TAS, a F&I, or a F&D operation (see 'Decoding RMW operations' in Figure 3.3) when a RMW operation is detected. In the MERASA processor this detection is triggered by the read and write signal in hardware both set to '1'. It is also possible to add further RMW operations by using the available bits in the data value to encode the corresponding RMW operations for being decoded in the augmented memory controller. For example, a F&A operation could be supported, however, parts of the identifier would then be needed to actually transfer the summand of the add operation. In detail, two bits of the 32 available bits could be used to decode the corresponding RMW operation, and the remaining 30 bits could be used for the summand of an F&A operation.

[3]The MERASA SystemC simulator and the MERASA FPGA prototype are deliverables of the EU project MERASA and are available for download at www.merasa.org [last accessed: April 2013]

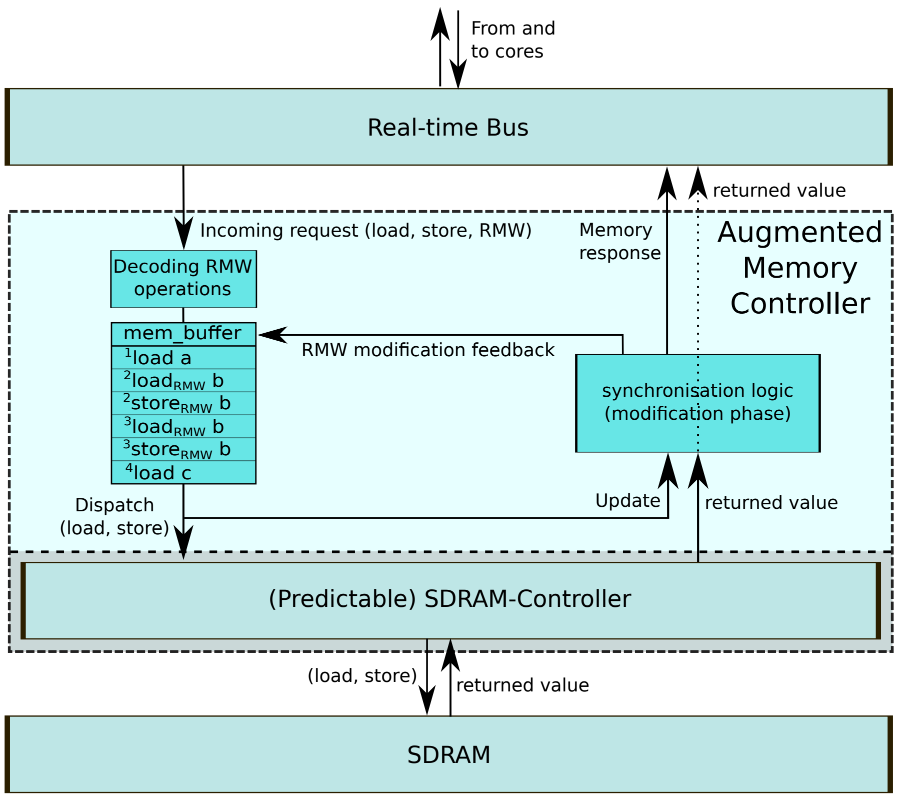


FIGURE 3.3.: Schematic overview of the augmented memory controller with implemented hardware support of RMW operations, and showing separation of augmented synchronisation logic from an SDRAM controller.

The advantage of this approach is that the generality of this implementation is not lost, as it is possible to reuse instructions of other ISAs for RMW operations in the memory controller as well. But, it must be assured that the compiler does not automatically generate the reused instruction used to encode/decode RMW operations. Otherwise it could get misinterpreted in the memory controller, as the data value that is used to decode the corresponding RMW operation might be arbitrary then. Also, the memory controller must be able to recognise such an instruction, however, in common ISAs for embedded systems mostly some unused or matching instructions are available for this modification (see related work in Section 3.2.4 for more details).

The shared memory interconnect (real-time bus in Figure 3.3) arbitrates memory requests in a round-robin fashion between the cores (see Paolieri et al. (2013, 2009a) and Ungerer et al. (2010) for details on different arbitration strategies in the MERASA processor). Memory accesses received over the bus are served on a FCFS basis, that is the augmented memory controller uses a FIFO queue for received accesses (*mem_buffer* in Figure 3.3), and dispatches the corresponding memory operations—either a load, a store, or a subsequent load and store of a RMW operation—to a (predictable) SDRAM controller of the shared memory.

The *synchronisation logic* in Figure 3.3 handles the modification phase of RMW operation by updating the value that is returned after the load phase and passes the new value to the corresponding store operation in the *mem_buffer*. If this modification concerns a constant value to be changed for the store operation (e.g. for TAS), the modification latency is hidden as it can be already started right after the load operation is dispatched. For RMW primitives that need to actually change the loaded value, that is incrementing/decrementing of F&I/F&D operations, an additional latency is added that defers the subsequent store operation. Depending on the hardware effort and memory frequency, it should be feasible to assume that such a modification can be done in one cycle (see also the settings for the timing analyses in Sections 4.1.3 and 4.2).

3.2.2. Read-Modify-Write Operations

To support software synchronisations in parallel programs, the hardware needs to provide atomic operations. In this section well known RMW operations (see Hennessy and Patterson 2003, Kruskal et al. 1988, Mellor-Crummey and Scott 1991a) like TAS and F&I/F&D are detailed, and the changes to their implementation in the augmented memory controller (see Figure 3.3) to be used in a HRT capable multi-core processor are explained. Also, LL/SC is discussed in Section 3.2.4, and it is presented how LL/SC instructions could be reused for the augmented memory controller with other multi-core platforms with ARM or PowerPC ISA. Table 3.1 shows an overview of the actual implemented RMW operations that are then presented in detail below.

Table 3.1.:
Overview of the implemented RMW operations in the augmented memory controller.

HW Primitive	Short Description
TAS	Loads a value and subsequently stores back a ’1’
F&I	Loads a value and subsequently stores back the incremented value
F&D	Loads a value and subsequently stores back the decremented value

Test-and-set

The TAS primitive is implemented in the augmented memory controller by first loading the lock value $val \in \{0, 1\}$ from memory address *address*. Subsequently, the memory controller stores a ’1’ at *address*. These operations are executed atomically in the augmented memory controller. Then, the loaded lock value *val* is returned to the thread issuing a TAS, where $val = 0$ signals that the lock is free and the thread got the lock, whereas $val = 1$ signals that the lock is held by another thread. The TAS operation is e.g. used for the TAS spin lock implementation presented in Section 3.3.1, and the mutex lock implementation in Section 3.3.3. Listing 3.1 shows the TAS implementation reusing the TriCore ISA’s `swap` instruction. The source code includes two different inline assembler code versions: one for the TASKING C compiler for TriCore, and one for the *tricore-gcc* by HighTec. Both only differ in the varying syntax for the use of inline assembler code. The value of variable *uint32_t data* is used as identifier in the augmented memory controller to execute a TAS operation, and also the read value from the memory will be returned to that variable, that is either ’0’ or ’1’.

LISTING 3.1:
Test-and-Set implementation in the MERASA RTOS using the *swap* instruction from the TriCore ISA

```
static inline uint32_t test-and-set(lock_t *lock) {
  uint32_t data = 1;
  #ifndef GCC //TASKING C compiler for TriCore (Altium Ltd.)
  __asm("swap.w %0,[%2]0":"=d"(data):"0"(r),"a"((uint32_t*)lock):"memory");
  #else // tricore-gcc compiler (HighTec EDV-Systeme GmbH)
  asm volatile ("swap.w [%2]0, %0":"=d"(data):"0"(data),"a"(lock):"memory");
  #endif
  return data;
}
```

Fetch-and-Increment/Fetch-and-Decrement

The F&I/F&D primitives are implemented slightly different in the MERASA processor as they typically are: the F&I implementation allows for specifying an upper limit to enable a cyclic counting behaviour. Therefore, a data word must be initialised in software before being used with F&I/F&D. That is the upper two bytes of the four byte memory word are used to store an upper limit value *lim*, whereas the lower two bytes are used for the actual *count* (should be typically initialised with '0'). At runtime, if e.g. the augmented memory controller recognises an F&I instruction by the *uint32_t data* variable coding (see Listing 3.2), it loads the four bytes word from memory address *address*. Then, the augmented memory controller sends the lower two bytes, the actual *count*, as fetched value to the core issuing the F&I instruction. Also, the implemented logic in the augmented memory controller checks if the lower two bytes are already higher than the initialised upper limit *lim* (the upper two bytes), i.e. if the counter *count* would overflow *lim* when being incremented. If this is not the case, *count* is incremented and stored back together with *lim*. Else, the counter *count* is set to '0'. This behaviour of the F&I operation is e.g. used in the ticket lock implementation (see Section 3.3.4), and for the cyclic counting to implement efficient FIFO queues (see Section 3.2.3).

LISTING 3.2:
Fetch-and-Increment implementation in the MERASA RTOS (see also Appendix A.1) reusing the *swap* instruction from the TriCore ISA

```
static inline uint32_t fetch_and_increment(uint32_t *address) {
  uint32_t data = 0x0000FFFF;
  #ifndef GCC //TASKING C compiler for TriCore (Altium Ltd.)
  __asm("swap.w %0,[%2]0":"=d"(data):"0"(data),
        "a"((uint32_t*)address):"memory");
  #else // tricore-gcc compiler (HighTec EDV-Systeme GmbH)
  asm volatile ("swap.w [%2]0,%0":"=d"(data):"0"(data),
                "a"(address):"memory");
  #endif
  return data & 0x0000FFFF;
}
```

The same holds for F&D (see Listing 3.3) that is recognised in the memory controller by a different *uint32_t data* coding then a F&I operation. If a F&D would decrement the loaded counter *count* so that *count* < 0, the lower two bytes of the store back value *count* are set to 0, and the upper two bytes are retained at the upper limit value *lim*. This cyclic counting allows to implement a busy-waiting spin lock with F&D (see Section 3.3.2). In that case F&D is executed in a loop, i.e. spins on the synchronisation variable at memory address *address*, until a value $\neq 0$ is returned. The F&D-based spin lock is used for the semaphore implementation presented in Section 3.3.5.

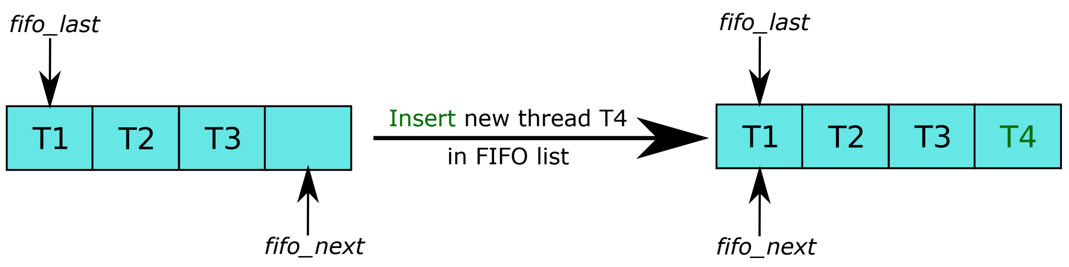
Listing 3.3:
Fetch-and-Decrement implementation in the MERASA RTOS (see also Appendix A.1) reusing the *swap* instruction from the TriCore ISA

```
static inline uint32_t fetch_and_decrement(uint32_t *address) {
  uint32_t data = 0x70007000;
  #ifndef GCC //TASKING C compiler for TriCore (Altium Ltd.)
  __asm("swap.w␣%0,[%2]0":"=d"(data):"0"(data),
        "a"((uint32_t*)address):"memory");
  #else // tricore-gcc compiler (HighTec EDV-Systeme GmbH)
  asm volatile ("swap.w␣[%2]0,%0":"=d"(data):"0"(data),
                "a"(address):"memory");
  #endif
  return data & 0x0000FFFF;
}
```

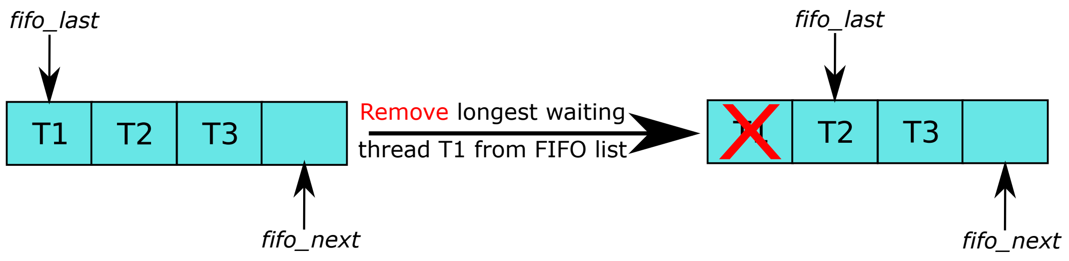
3.2.3. FIFO Queue with F&I

The F&I primitive implementation, proposed in this thesis, can be employed for a concurrent FIFO queue. In that case, F&I is used to increment the index for the insert/remove operation of threads in a FIFO buffer. The only requirement is that the upper limit of the FIFO queue needs to be known a-priori by the programmer, that is the upper bound needs to be set for the F&I primitive. As in each case two bytes are used for the counter and the upper limit, it is possible to count up from 0 to $2^{16}-1$. With the chosen implementation of F&I, it is then easily possible to manage a FIFO queue applying the implemented cyclic counting of F&I.

In detail, the F&I operation is used to increment the index atomically, and, if the last index is reached, the next fetched index is ’0’ again, as the upper limit for F&I is reached. This allows, for instance, to implement a FIFO ordered waiting list for HRT threads. In this example (cf. Figure 3.4), each HRT thread is represented by its unique *thread id*, and inserting a HRT thread in the waiting list means inserting the corresponding *thread id*. Then, insertion of HRT *thread ids* into the waiting list increments the index *fifo_next* for the next insert location, and removing a HRT thread from the waiting list increments the next remove location *fifo_last* in the waiting list (see Figure 3.4). Overwriting of *thread ids* in the FIFO queue is not possible, as the number of HRT threads is known at design time, e.g. restricted by the number of cores in a multi-core processor, and therefore the upper limit for the F&I primitive can be adjusted accordingly.



(a) Insert operation



(b) Remove operation

FIGURE 3.4.: Insert(a) and remove(b) operations of threads with the FIFO queue using F&I to change the position of pointers *fifo_next* respectively *fifo_last*.

The advantage of this approach is that, contrarily to a fully software managed linked list, the list access does not need to be secured with a critical section as the access to it is done with an atomic RMW operation, that is F&I. Therefore, computation and waiting times in the WCET are possibly reduced (see evaluation results in Section 4.3). The FIFO queue with F&I is used, for example, to manage the waiting list in the semaphore implementation in Section 3.3.5, and also in the F&I barrier implementation detailed in Section 3.3.6.

3.2.4. Related Work

ARM proposes local and global monitors[4] for their LL/SC implementation. The monitors keep track of on-going load-linked operations, so-called reservations (the load-linked instruction is called `ldrex`, a store-conditional instruction `strex`). While the local monitor observes and manages reservations for local memories, the global monitor does the same at a global shared memory. There are several limitations on when and how reservations must be cancelled, e.g. for a context switch, or for memory operations (esp. store operations) that access other, near memory locations. The term near stems from the Exclusives Reservation Granule (ERG), which stipulates a minimum size of the reserved (tagged) memory region; the size for ERG is between 8-2048 bytes. That is a store operation in the range of ERG leads to unpredictable behaviour[4].

[4]see *ARM Synchronization Primitives Development Article* from `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/index.html`). [last retrieved: April 2013]

Also, only one reservation per processor in a multiprocessor system is allowed in the global memory, as well as in the local memory. Hence, further reservations from different threads calling `ldrex` cancel any previous reservations. For context switches, it is necessary to invalidate reservations from `ldrex` by calling a `clrex` instruction (for the ARMv6-M ISA (2010), which does not provide a `clrex` instruction, this needs to be done with a dummy `strex` instruction). Please note that for non-cache-coherent ARM processor implementations, the `ldrex`/`strex` instructions bypass the cache and reservations are only done in the global monitor.

The PowerPC ISA (2010) features similar primitives for LL/SC, that is `lwarx` and `stwcx` (also LL/SC pairs for operations on bytes, half-words, and double-words are provided). Just like the ARM LL/SC primitives, the PowerPC ISA (2010) allows only one reservation per processor, and the granularity (called reservation granule) is implementation dependent, but must be of a minimum size of 16 bytes (2^n bytes with $n \geq 4$).

Several papers, e.g. from Valois (1995) and Engdahl and Chung (2010), evaluated the average-case performance using LL/SC primitives in multi-core processors for synchronising data exchange. But, a major drawback of LL/SC implementations in HRT systems is the problem of bounding the waiting time for the conditional store operation. So far, only highly overestimated upper bounds, if any at all, can be computed, when LL/SC is used as a hardware synchronisation primitive. This stems from the problem of limited numbers of reservation, and especially from the conditional nature of the store operation. In the most multi-core architectures it will be difficult or infeasible for static timing analyses to find a (preferably tight) upper bound for conditional-store operations.

However, similar to the reusing of the `SWAP` operation of the TriCore ISA (2008) in the MERASA processor the `ldrex` of ARM processor ISAs could be reused to implement the synchronisation logic in a memory controller. The same holds for the similar instruction `lwarx` of PowerPC ISA (2010). This is e.g. done in the EU-project parMERASA[5], in which PowerPC cores are employed, and the `lwarx` instruction is reused to implement the needed logic for F&A primitives in a memory controller.

A further RMW operation that might be offered more frequently in future multi-core architectures is the CAS primitive. Especially the use in non-blocking algorithms seems promising (see Section 3.3.7). But, even if such a primitive is not implemented, it is possible to emulate a CAS primitive with LL/SC operations.

The cyclic FIFO buffer, presented in Section 3.2.3, features a similar behaviour as the *two-pointer FIFO queue* presented by Valvano (2011). Valvano (2011) presents the FIFO queue solution for decoupling data exchange between producers and consumers. However, the advantage of the novel managing of pointers in the FIFO implementation with atomic RMW operations, in that case with the cyclic counting of the F&I primitive presented in Section 3.2.3, is the possibility of manipulating the pointers concurrently, without the need of a critical section to secure the access.

[5]see the *Multi-Core Execution of parallelised Hard Real-Time Applications Supporting Analysability* (parMERASA) project website at www.parmerasa.eu [last accessed: April 2013]

3.3. Software Synchronisation Techniques

Software synchronisation techniques presented in this section are part of an RTOS (see Gerdes et al. 2012b) that extends the RTOS presented by Wolf et al. (2010a, 2011, 2010b) (see also Appendix A.1). In the following, the implemented synchronisation constructs are presented, that is *F&I/F&D spin locks*, *ticket locks*, *semaphores*, and *software barriers* that are using the F&I/F&D primitives, as well as synchronisation techniques based on the TAS primitive, namely a *TAS spin lock* implementation, and a *mutex lock* implementation according to POSIX (2008). Table 3.2 gives an overview of the implemented HRTcapable software synchronisation techniques. The implemented software synchronisations have been carefully designed to be HRT capable in a shared-memory multi-core processor.

As described in Chapter 2, synchronisation methods can be separated into two different categories, that is either blocking or busy-waiting (see Table 2.2). Busy-waiting means that the thread executes the synchronisation function in a loop until the lock is gained, whereas blocking means that a thread tries to get the lock and is suspended if not succeeding. On the one hand, blocking synchronisation methods are handy, as busy-waiting algorithms can consume a lot of processor time and add contention on the memory system (and other shared resources which need to be taken into account when accessing a synchronisation variable). Every non-successful try to gain the lock when busy-waiting should be avoided for average-case performance. With suspension it is possible to avoid unnecessary accesses, but suspending and waking up takes longer than busy-waiting depending e.g. on the hardware. From the view point of static timing analysis, suspension does not really help. In the real execution path a thread might be suspended, but, for the worst-case path analysis, it needs to be assumed that it is not. Therefore, it highly depends on the parallelised HRT programs and length of critical section that is guarded, if and which busy-waiting or blocking software synchronisation method should be used (see Gerdes et al. 2012b).

Table 3.2.:
Overview of the implemented software synchronisation techniques for the HRT capable MERASA multi-core processor.

Software Synchronisation Technique	Used RMW Operations
Spin lock (Section 3.3.1)	TAS
F&I/F&D spin lock (Section 3.3.2)	F&I/F&D
Mutex lock (Section 3.3.3)	TAS
Ticket lock (Section 3.3.4)	F&I
Semaphore (Section 3.3.5)	F&I/F&D
Software Barrier (Section 3.3.6)	F&I

Synchronisation techniques need to fulfil specific requirements for being HRT capable (see Section 2.2.1). Concerning *progress*, strong fairness between threads that are competing to access a critical section needs to be achieved. In the following it is shown that not all typical implementations, e.g. barriers with conditionals, or spin locks respectively mutex locks using the TAS primitive, are strictly fair (see Sections 3.3.1, 3.3.3, and 3.3.6). Hence, fair progress between competing threads and an *upper bounded waiting time* for critical sections, that is as low as possible, are both needed to enable a static WCET analysis at all, respectively to achieve tight WCET guarantees. Chapter 4 presents a comparison of the implemented synchronisations and the gain in the WCET guarantees of parallelised HRT programs using these software synchronisations.

3.3.1. TAS Spin Locks

Algorithm 3.1 shows a simple, commonly used spin lock implementation with the TAS primitive. The implementation is busy-waiting, that is all threads that are competing for a lock issue TAS instructions until they gain the lock.

Algorithm 3.1 Spin locks with TAS

```
Spin lock() function
1: //Enter critical section
2: while !(test-and-set(addr,reg)) do
3:     /* test-and-set: Stores '1' at memory address addr and loads value from addr to
       register reg */
4: end while
5: //Remainder critical section
6: ...
Spin unlock() function
1: //Leave critical section
2: store(addr,0) // respectively test-and-set(addr,1)
```

This implementation is possibly the simplest synchronisation function to secure a critical section, and it is widely used. Lately, other synchronisation functions like ticket locks, e.g. for the Linux Kernel, and queuing spin locks for real-time systems are used, because of the lack of fairness in spin locks (see Sections 3.3.4, and 3.3.7). Please note that the above implementation of a TAS spin lock does not fulfil the requirement of progress, namely fair progress, for each thread without specific arbitration for the shared memory. Thus, the computation of a WCET guarantees might not be possible at all, or introducing a large overestimation if no real-time aware arbitration for the shared resources is used. However, in the MERASA multi-core processor fairness of TAS spin locks is assured by the arbitration strategy in the real-time bus, as presented by Paolieri et al. (2013, 2009a). Nonetheless, using the above TAS spin lock implementation with other memory interconnects that arbitrate memory requests in a fair manner might comply real-time requirements (see related work presented in Section 4.5).

Also, if the computation time between subsequent locking of one thread is very short, that is for instance there is no computation in between, the fairness condition could be violated. This behaviour would be similar to the reinitialisation problem of barriers (see Section 3.3.6), but instead of causing a deadlock, it would hinder other threads to enter the critical section secured with a TAS spin lock, which e.g. motivated the use of ticket locks in the Linux Kernel (see also Section 3.3.7). In some implementations, the *store* operation used for the *unlock()* function in Algorithm 3.1 is substituted by a `swap` or a different RMW operation, i.e. a TAS or F&I operation. In that way, it is easier to recognise that access as an access to a synchronisation variable in the static WCET analysis. However, the WCML of a *store* is lower than for a RMW operation, and from the functional point of view a *store* is sufficient to unlock the spin lock. Though, it is mandatory that the memory address *addr* is then marked or automatically detected as a synchronisation variable for the static WCET analysis tool (see details in Chapter 4).

3.3.2. F&I/F&D Spin Locks

Another busy-waiting spin lock implementation can be build, as stated below in Algorithm 3.2, with the F&I/F&D RMW operations in the MERASA processor with the augmented memory controller. It is very similar to the above presented TAS spin lock, and also shares the same requirements for fair progress.

Algorithm 3.2 Spin locks with F&I/F&D

F&I/F&D Spin *lock()* function

```
1: //Enter critical section
2: while fetch-and-decrement(addr) do
3:
4: end while
5: //Remainder critical section
6: ...
```

F&I/F&D Spin *unlock()* function

```
1: //Leave critical section
2: fetch-and-increment(addr) // respectively store(addr,1)
```

The semantic of the F&D operation allows to execute it repetitive, even if the value is already ’0’, because the implemented F&D operation does not decrement below ’0’ (see Section 3.2.2). The F&D operation can then be used for a busy-waiting spin lock, and spins until a value > 0 is fetched (see line 2 in Algorithm 3.2). The *unlock* operation uses a F&I operation to signal that the lock is free again, that is writing a 1. This could also be done by a simple *store*, however, as the F&I/F&D operations are executed on a specific representation of data in a 32 bits data word–storing the limit in the first 16 bits and the actual counter value in the last 16 bits–a F&I operation automatically satisfies this representation by its implementation in the MERASA RTOS and augmented memory controller. The F&I/F&D spin lock is used in the semaphore implementation detailed in Section 3.3.5.

3.3.3. Mutex Locks

The HRT capable mutex lock is a blocking synchronisation function which has been implemented compliant to POSIX (2008). It uses a TAS spin lock for the critical sections inside the mutex lock, and a software based FIFO waiting list to guarantee fairness between concurrent threads. In detail, acquiring respectively releasing a mutex lock needs to acquire the TAS spin lock on *mutex→guard* which secures the critical region for accessing the global variable *mutex→the_lock*. Suspension and waking up of threads is handled from a software managed linked list (see Algorithm 3.3) secured by the same TAS spin lock on *mutex→guard*.

Algorithm 3.3 Blocking Implementation of Mutex locks with TAS Spin Locks

Mutex *lock()* function

```
1: //Enter critical section
2: while spin_lock(&mutex→guard) do
3:
4: end while
5: if mutex→the_lock $\neq$ 0 then
6:     Enter waiting list
7:     spin_unlock(&mutex→guard)
8:     Suspend thread
9: else
10:    mutex→the_lock = 1
11:    spin_unlock(&mutex→guard)
12: end if
13: Set as mutex owner
14: //Remainder critical section
15: ...
```

Mutex *unlock()* function

```
1: //Leave critical section
2: while spin lock not acquired do
3:     spin_lock(&mutex→guard)
4: end while
5: if thread in waiting list then
6:     Wake thread
7:     Unset as mutex owner
8: else
9:     mutex→the_lock = 0
10: end if
11: spin_unlock(&mutex→guard)
```

To acquire a mutex lock all concurrent HRT threads compete for the TAS spin lock to enter the critical region (line 2 of lock() function in Algorithm 3.3). A HRT thread that successfully enters this critical region, checks if the global variable *mutex→the_lock* indicates a free mutex lock (*mutex→the_lock* == 0) or a taken mutex lock (*mutex→the_lock* == 1). If the mutex lock is free, the HRT thread sets *mutex→the_lock* to '1', and releases the TAS spin lock (lines 10,11); this HRT thread now holds the mutex lock—it is the *mutex owner*. Other competing HRT threads, on gaining the TAS spin lock inside the mutex lock, check the global variable *mutex→the_lock* which is now set to '1', and therefore enter a linked list before suspending. The linked list contains all waiting HRT threads in FIFO-order to guarantee fairness, i.e. the longest waiting thread wakes up and acquires the mutex lock when the previous lock holding HRT thread releases it. Finally, the HRT thread releases the TAS spin lock inside the mutex lock and suspends.

When a HRT thread releases a mutex, it busy-waits on gaining the TAS spin lock for the critical region in which it wakes the longest waiting HRT thread from the linked list. To ensure that the woken HRT thread acquires the mutex lock, and not other HRT threads which are not suspended yet and that are competing for the mutex lock, the active mutex lock holder does not set *mutex→the_lock* to '0' after waking up a waiting HRT thread. Instead, it releases the TAS spin lock, and thus, if a HRT thread, other than the woken HRT thread, acquires the TAS spin lock inside the mutex lock, it finds *mutex→the_lock* still set to '1' and suspends. The woken HRT thread does not need to check *mutex→the_lock* again, and hence acquires the lock becoming the *mutex owner*. Taking into account the FIFO-order in the linked waiting list as well, progress of HRT threads for the mutex lock implementation using a real-time aware memory bus arbitration—needed for fairness of the TAS spin lock implementation (see Section 3.3.1)—can be assured.

3.3.4. Ticket Locks

Algorithm 3.4 Busy-Waiting Implementation of Ticket locks with F&I

```
1: //Enter critical section
2: my_ticket = F&I(ticket_id)
3: while my_ticket ≠ now_served do
4:
5: end while
6: //Remainder critical section
7: ...
8: //Leave critical section
9: F&I(now_served)
```

The semantic of *ticket locks* (Mellor-Crummey and Scott 1991a), based on Lamport's bakery algorithm, is as follows: each HRT thread gets a unique *ticket_id* when trying to access a critical region (line 2 in Algorithm 3.4). HRT threads are allowed to enter the critical region when their *ticket_id* matches the current value of *now_served* (line 3).

The threads are busy-waiting, until their ticket id *my_ticket* matches the value of *now_served*. After a thread leaves a critical section, it increments *now_served* (line 9), and the thread with the appropriate *ticket_id* can now enter the critical section.

The atomic incrementing of *ticket id* and *now_served id* is done with the F&I primitive in our implementation (see Algorithm 3.4). Thus, ticket locks implement a busy-waiting spin lock, which is, contrary to the above presented TAS spin locks, fair, independently of the arbitration strategy in the memory interconnect. For instance, if a memory bus arbitrates memory request in a non-round-robin fashion, e.g. randomly, the following request pattern is possible for a dual-core processor featuring two HRT threads: {Thread 1(T_1), T_1, T_1, Thread 2(T_2), T_1, T_1, T_1, T_1, T_2, T_1, ...}. So, thread T_2 is only allowed to access the global memory only very infrequently, whereas thread T_1 gains access very often. Now, the critical section is guarded by a TAS spin lock in one case, and by a ticket lock in another case. In the first case with a simple TAS lock, thread T_1 might hold the lock every time a request to gain the lock from thread T_2 is dispatched to the memory controller. Also, thread T_1 tries to regain the lock just after it released it. So, thread T_2 might never gain the lock guarding the critical section, or, only very rarely. Hence, a non-fair distribution of accesses to the critical section is possible.

TABLE 3.3.:
Example of two threads T_1 and T_2 and their local memory state versus global memory state when competing for a critical section secured with ticket locks.

	thread-local registers		global shared memory	
Action	*my_ticket* T_1	*my_ticket* T_2	*ticket_id*	*now_served*
init	-	-	0	0
T_1 try acquire T_1 acquire *successful*	0	-	1	0
T_2 try acquire T_2 acquire ***not** successful*	0	1	2	0
T_1 release	0	1	2	1
T_1 try acquire T_1 acquire ***not** successful*	2	1	3	1
T_1 try acquire T_1 acquire ***not** successful*	2	1	3	1
T_2 acquire *successful*				
T_1 try acquire T_1 acquire ***not** successful*	2	1	3	1
T_2 release	2	1	3	2
...				

Table 3.3 depicts the latter case, assuming a ticket lock guarding the critical section. Even if the requests of thread T_2 to gain the lock are only dispatched infrequently, it eventually gets a ticket *my_ticket*, namely each time the thread is dispatched while not holding a ticket yet. So, the very frequently dispatched thread T_1 finds his ticket id *my_ticket* being higher than the actual *now_served* value after thread T_2 has been able to access the global memory. In summary, ticket locks presume starvation-freedom for very infrequently dispatched threads. However, one might argue that an equally fair behaviour of TAS locks might be enforced by using backoff algorithms. A discussion on the applicability of locks with backoff in HRT systems is done in Section 4.3.1.

For a static timing analysis, the waiting times for threads until successfully gaining a lock need to be upper bounded. In the above mentioned example, this might not be possible. Though, if the dispatching of requests to the memory is not strongly fair, meaning each thread is dispatched at the same frequency, but weakly fair, and an upper bound delay for the infrequently dispatched thread is known, a static WCET analysis using ticket locks is possible, whereas using TAS spin locks it is not.

3.3.5. Semaphores

The HRT capable semaphores are implemented according to POSIX (2008), but using the F&I/F&D primitive. In POSIX (2008), the original *P-operation* and *V-operation* from Dijkstra (1968) are being referred to as *wait()* and *post()*, which are also used in the following. Semaphores are part of the *Real-time extensions* (IEEE Std 1003.1b-1993) of POSIX (2008), and are used with the prefix *sem_* instead of *pthread_* (see Nichols et al. 1996, p. 54).

The proposed HRT capable semaphore implementation uses a struct *sem* with the following members:

- $sem \rightarrow value$: semaphore counter
- $sem \rightarrow waitlist\_lock$: binary lock for the waiting list
- $sem \rightarrow waitlist\_fifo[]$: FIFO buffer for the waiting HRT threads
- $sem \rightarrow waitlist\_entries$: counter for the number of HRT threads waiting

Algorithm 3.5 depicts the *wait()* and *post()* method. The *wait()* method first needs to check if the resource secured by the semaphore is free. This is done by atomically fetching and decrementing the semaphore counter $sem \rightarrow value$ (line 2 in *wait()* function in Algorithm 3.5). A resource is successfully acquired if a value ≥ 0 is fetched, otherwise the thread enters a waiting list and suspends.

Inserting and removing a thread from the waiting list is secured in a critical section (lines 3-11 in *wait()* respectively lines 2-7 in *post()* function). This is needed as otherwise a thread freeing a semaphore might conflict with a thread that is currently trying to enter the waiting list. Contrarily to the mutex lock implementation that secures the waiting list with a TAS spin lock (see Section 3.3.3), the semaphore implementation uses a F&I/F&D spin lock (see Section 3.3.2) in lines 3-5 of the *wait()* function, respectively in lines 2-4 of the *post()* function in Algorithm 3.5.

Algorithm 3.5 Semaphores with F&I/F&D (Blocking)

Semaphores *wait()* function ≡ $P()$

```
1: //Enter critical section
2: if F&D(&sem→value) ≤ 0 then
3:    while !F&D(&sem→waitlist_lock) do
4:
5:    end while
6:    if F&D(&sem→value) ≤ 0 then
7:       //Add to waitlist_fifo
8:       sem→waitlist_fifo[F&I(&sem→fifo_next)] =
         thread_to_suspend
9:       F&I(&sem→waitlist_entries)
10:      F&I(&sem→waitlist_lock)
11:      suspend()
12:      //After wakeup
13:      F&I(&sem→waitlist_lock)
14:   else
15:      F&I(&sem→waitlist_lock)
16:   end if
17: end if
18: //Remainder critical section
19: ...
```

Semaphores *post()* function ≡ $V()$

```
1: //Leave critical section
2: while !F&D(&sem→waitlist_lock) do
3:
4: end while
5: if F&D(&sem→waitlist_entries) then
6:    //Unsuspend longest waiting thread from waitlist_fifo
7:    thread_to_unsuspend =
      sem→waitlist_fifo[F&I(&sem→fifo_last)]
8: else
9:    F&I(&sem→value)
10:   F&I(&sem→waitlist_lock)
11: end if
```

Furthermore, it is important that a thread leaving the semaphore, that is executing the *sem_post()* method, wakes one of the potentially waiting threads that is already suspended and waiting for the semaphore (lines 5-7 in *post()* function), as otherwise a waiting thread could starve. Additionally, the problem that a thread leaving the semaphore executes the critical section before a thread competing for that resource needs to be solved. Otherwise, this could also lead to starvation of the latter one. Therefore, an additional F&D primitive to check if the resource was freed while the thread was busy-waiting to access the critical section needs to be added (line 6 in *wait()* function). The suspended threads are managed in a FIFO queue (line 8 in *post()* respectively line 7 in *wait()* function) that uses the cyclic implementation of the F&I operation as detailed in Section 3.2.3.

A binary semaphore, that is a semaphore that only allows the binary values '0' and '1', has the same functionality as a mutex lock with the difference that the semaphore implementation does not use the concept of an owner contrarily to the mutex lock implementation. Beside that, this allows to compare the WCET of waiting times in mutual exclusion implementations with different RMW operations, namely TAS for mutex locks and F&I/F&D for binary semaphores, detailed in Chapter 4.

Beside the blocking semaphore implementation, also a busy-waiting semaphore without a waiting list has been implemented, but is not further detailed in this thesis.

3.3.6. Software Barriers

Simple software implementations for barrier synchronisation could lead to deadlocks and hindering overall progress. They are often prone to the so-called *reinitialisation* problem, that is a thread leaving the barrier after successfully resetting the barrier condition might reenter it immediately again. This could lead to the problem that the reentering threads resets the barrier condition, and hence other threads that did not leave the barrier by now could still spin on the now changed barrier condition: a deadlock would be caused. Hennessy and Patterson (2003) describe such a simple barrier implementation and behaviour with the Algorithm 3.6 presented below.

Algorithm 3.6 Simple Busy-Waiting Barrier from Hennessy and Patterson (2003)

```
 1: lock (counterlock); // count arriving threads in critical section
 2: if count == 0 then
 3:     release = 0; // first thread: reset release
 4: end if
 5: count = count + 1; // count arrived threads
 6: unlock (counterlock);
 7: if (count==total) // all threads arrived then
 8:     count = 0; // reset counter
 9:     release = 1; // release all threads
10: else
11:     spin (release==1); // wait for total number of threads to arrive
12: end if
```

In the Algorithm 3.6 *lock()* and *unlock()* should provide a basic spin lock mechanism, and *count* counts the number of threads that reached the barrier, whereas *total* is the total number of threads that need to reach the barrier before all threads are allowed to continue. The major drawback of that implementation is, as stated by Hennessy and Patterson (2003), the spinning of threads in line 11 (*spin(release==1);*. For example, assume the program example in Listing 3.4. If a number of threads execute the *parallel_code_function()*, they enter the barrier at *enter_barrier* in a loop. The time it takes for each thread to reach the barrier again depends on the *condition*, that is the code they execute depending on that condition: either *do_heavy_work()* or *do_nothing()*. Now, Hennessy and Patterson (2003) state that in that kind of example, a thread T_1 might reenter the barrier before all other threads have left it. They especially stress that this could happen if a thread T_2, currently spinning on the *release* value (line 11 in Algorithm 3.6), is swapped out, and before that thread T_2 is swapped in again, another thread T_1 enters that barrier again and resets the *release* value (line 3 in Algorithm 3.6). For instance, when the condition of thread T_1 changes from *true* to *false* in the code example in Listing 3.4. Or, if for some other reason the code section between two executions of the same barrier is very short. In any way, if that would happen, the thread T_2 is delayed infinitely, because it spins on the meanwhile changed value of *release*. Even more severe, if that barrier is executed in a loop, the whole program cannot progress any more, as the *total* value of threads that is needed to successfully pass the barrier cannot be achieved anymore when the thread T_2 does not progress.

Listing 3.4:
Parallel Program Example for the Simple Barrier Implementation in Algorithm 3.6

```
void *parallel_code_function(void)
{
...
  while(true)
  {
    if(condition)
    {
      // this code section will consume a lot of processing time
      do_heavy_work();
    }
    else
    {
      // this code section will consume hardly any processing time
      do_nothing();
    }
    // all threads enter this barrier in a loop
    enter_barrier();
    ...
    // changes the condition depending on the thread ID
    change(threadID, condition);
  }
...
}
```

A possible solution of also counting all threads when leaving the barrier overcomes that problem, but, according to Hennessy and Patterson (2003), with huge costs on latency and contention. Other solutions that achieve better scaling are e.g. *sense-reversing barriers* (see Hennessy and Patterson 2003, Hensgen et al. 1988), but they do not provide a bounded number of remote operations. But, for being HRT capable, a possible barrier implementation needs to provide bounded waiting times, and those are equated with bounded operations.

For the case of blocking barriers, a commonly implemented version uses mutex locks and conditional variables. Algorithm 3.7 shows a pseudo code for such an implementation. The problem with those implementations, despite that it is deadlock free, is possible overestimation in static timing analyses. The reason for this is, as in the above simple barrier implementation, the possible reentering of a thread at the barrier while another thread is still suspended at that barrier. In detail, this behaviour is based on the conditional variable (line 9 in Algorithm 3.7) that shares the same mutex lock *barrier→mutex* that secures the critical section when entering/leaving the barrier (line 2/12). Also, that implementation does not scale well, because of contention on that mutex lock from threads entering/leaving the barrier. Thus, such an implementation is not recommended in highly contended systems, but as well not in HRT systems.

Algorithm 3.7 Blocking Barrier with Conditional Variable

```
 1: // Enter critical section
 2: mutex_lock(&barrier→mutex)
 3: barrier→called++;
 4: if (barrier→called == barrier→needed) then
 5:     barrier→called = 0;
 6:     conditional_broadcast(&barrier→cond); // wake all waiting threads
 7: else
 8:     // enter waitlist, suspend and leave critical section
 9:     conditional_wait(&barrier_cond→, &barrier→mutex);
10: end if
11: // Leave critical section
12: mutex_unlock(&barrier→mutex);
```

A better solution than barriers with conditionals is to use subbarriers introduced by Marejka (1994), and e.g. used in the Legion SPARC simulator[6]. In that implementation the competition between threads at a barrier in different phases of the parallel program is solved by switching from one subbarrier to another when all needed threads have reached the barrier. Thus, the leaving threads are exiting one subbarrier whereas other threads, which might again execute the barrier code, enter the other subbarrier. The major goal of the subbarrier implementation is to scale well, but it is also applicable for parallel HRT programs.

[6]see `http://kenai.com/projects/legion/sources/legion-opensparc/content/src/generic/barrier.c` for the source code in the LEGION SPARC simulator. [last retrieved: April 2013]

In Section 4.3.3 static WCET analysis results of the subbarrier implementation (and WCET guarantees of parallel HRT programs using that implementation) are shown and compared to the following recommended implementation with the F&I primitive.

The recommended solution for barriers in parallel HRT programs is to implement barriers with the F&I primitive. This blocking barrier implementation, shown in Algorithm 3.8, uses a waiting list for suspended threads as described for the FIFO queue with F&I in Section 3.2.3. Using F&I for barriers is a well known concept, e.g. shown by Hennessy and Patterson (2003), and also overcomes the static timing analyses' overestimation of the implementation of barriers with conditional variables. For the barrier implementation with F&I the following struct is used:

- *barrier* $\rightarrow$ *needed*: number of threads needed at the barrier to continue
- *barrier* $\rightarrow$ *runners*: counts the number of threads currently waiting at the barrier
- *barrier* $\rightarrow$ *waitlist_lock*: lock that secures the waiting list of suspended threads
- *barrier* $\rightarrow$ *waitlist*[]: waiting list for the suspended threads

Algorithm 3.8 Barriers with F&I

Barrier *wait()* function

```
 1: //Enter critical section
 2: while !F&D(&barrier→waitlist_lock) do
 3:
 4: end while
 5: cur_runner = F&I(&barrier→runners);
 6: if (cur_runner ≥ barrier→needed - 1) then
 7:     //Last thread reaches the barrier
 8:     Wakes all waiting threads
 9:     for i = 1 → (barrier_needed - 1) do
10:         unsuspend(waitlist_fifo[i]);
11:     end for
12: else
13:     // Enter waitlist, suspend and leave critical section
14:     barrier→waitlist_fifo[cur_runner] = my_threadID;
15:     F&I(&barrier→ waitlist_lock)
16:     return
17: end if
18: //Leave critical section (only last thread executes this)
19: F&I(&barrier→ waitlist_lock)
```

Algorithm 3.8 depicts the behaviour of the F&I barrier implementation. All threads that enter the barrier are suspended, as long as the needed number of threads is not reached (lines 5, 8-9 in Algorithm 3.8). The waiting list for threads is organised as for the blocking semaphores with a FIFO buffer managed with F&I (see Section 3.2.3). When the last needed thread enters the barrier, it wakes all waiting threads from the waiting list *barrier→waitlist[]*, and all threads continue their execution (lines 9,10). Then, only the last thread unlocks *barrier→waitlist_lock*. Threads that try to reenter the barrier in the next iteration are busy-waiting at the *barrier→waitlist_lock* until that last thread of the previous iteration increments *barrier→waitlist_lock* and leaves the barrier (lines 2, 18 in Algorithm 3.8).

The advantage of this approach is that busy-waiting of reentering threads is not relevant for the WCET, as the WCET of a code section with barriers depends on the last thread arriving at the barrier. But, with the above proposed F&I barrier implementation, only the first thread that reenters the barrier is affected. However, this implementation is specifically engineered for parallel HRT programs with a shared-memory programming model, and for worst-case performance. That is, the implementation might not scale well for a high number of threads (>> 4) and is not recommended when average-case performance in NHRT systems is the main goal, because of possible contention. More details on the WCET analysis of this barrier implementation are in Section 4.3.3, and related busy-waiting and blocking barrier implementations are shown in Section 3.3.7.

3.3.7. Related Work

In the following, related and relevant work on locks, barriers, non-blocking synchronisation techniques, and transactional memory for real-time systems and multiprocessors respectively multi-core processors is discussed in detail.

Locks

Molesky et al. (1990) present an arbitration for a bus, the *Deferred Bus* theorem, which is the baseline for the bus arbitration that is used in this thesis to assure fairness of spin locks (see Paolieri et al. 2009a). Molesky et al. (1990) show that their *Deferred Bus* enables synchronisation mechanisms for mutual exclusion with linear waiting, and bounded semaphores for predictable synchronisation in multiprocessor systems. Though, the use of ticket locks is more flexible concerning the bus arbitration, which is also the reason why ticket locks are used in the Linux Kernel as a fair spin lock mechanism[7].

Lubachevsky (1984) introduced software synchronisations, among others semaphores, which are build on the F&A primitive of *ultracomputers* (see Schwartz 1980). The proposed and correctness-proven busy-waiting semaphore with F&A is similar to the blocking implementation discussed in Section 3.3.5.

[7]more details are to be found as patch notes of Nick Piggin on ticket locks in the Linux Kernel at `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=314cdbefd1fd0a7acf3780e9628465b77ea6a836` [last retrieved: April 2013]

Anderson (1990) introduces queuing spin locks using unique ids in shared-memory multiprocessors with cache coherence. The approach is similar to ticket locks established by Mellor-Crummey and Scott (1991a) despite that the ticket locks are implemented with the F&I primitive. However, both papers do not feature real-time issues, but it is the key aspect in the implementation of ticket locks with the F&I primitive proposed in this thesis. That is the focus on ticket locks in this thesis is on assuring strong fairness between HRT threads without requiring a specific bus arbitration.

Graunke and Thakkar (1990) present results on busy-waiting locks in a shared memory multiprocessor system with cache coherence in the high-performance domain. They analyse the impact of different busy-waiting lock implementations on contention and average-case performance. Graunke and Thakkar (1990) recommend to not use simple TAS locks for high contended synchronisations, but using queuing spin locks in that case. Graunke and Thakkar (1990) also advocate to not use TAS locks in multiprocessor systems with cache coherence, as they show a poor average-case performance due to adding a lot of contention. Furthermore, they show that using backoff algorithms, or delays as they introduce it, reduces contention and increases the average-case performance. However, the use of backoff algorithms for busy-waiting locks does not spark any gain in the static timing analysis applied in this thesis (see Section 4.3.1).

Further spin lock implementations, which might allow timing predictability in shared-memory multiprocessors, like MCS locks (see Mellor-Crummey and Scott 1991a) or CLH locks (see Craig 1993, Scott and Scherer 2001), require a cache coherence protocol or, for MCS locks, a complex allocation and pointer arithmetic in local memory on non-cache-coherent systems. However, the HRT capable MERASA multi-core processor (see Ungerer et al. 2010) used in this thesis does not employ a cache coherence protocol, because cache coherence protocols need complex hardware and/or software solutions, which hinder a static WCET analysis or even render it impossible (see Schoeberl and Puschner 2009). Also, the case for queued spin locks is to reduce the overhead and contention of busy-waiting synchronisation primitives to improve the average-case execution time. But, for a tight WCET analysis, the focus is on reducing the WCET overestimation and pessimism introduced from (slower) RMW operations on shared memory. Moreover, the above mentioned queuing spin lock implementations also use RMW operation on the shared memory, i.e. a CAS operation, and therefore would still add pessimism for concurrently running threads in the WCET analysis (see Section 4.3.4 for more details).

Barriers

Barriers are a well-known concept from high-performance computing. Contrarily to locks, which can be seen as *memory barriers*, barriers are more general used for progress coordination. Barriers can be categorised into three classed: centralised, decentralised, and hierarchical barriers (cf. Sartori and Kumar 2010).

Centralised barriers synchronise on a global counter. This technique does not scale well for very larger parallel programs, however, for the case of relatively small shared-memory multi-core processors, e.g. as the MERASA processor used in this thesis is, the worst-case efficiency is sufficient and they are analysable with static WCET tools.

Centralised barriers may also perform better than decentralised or hierarchical barriers, as e.g. shown for multiprogrammed workload by Markatos et al. (1991). Also, the analysis of parallelised HRT programs, which were implemented for the MERASA multi-core processor, show that the portion of barriers is rather low in comparison to the residuary code (see Section 4.3.3). For those reasons, in this thesis HRT capable barriers were only implemented as centralised barriers in software with hardware support, e.g. specific RMW operations, but not purely in hardware, that is no hard-wired barriers as, e.g., proposed by Shang and Hwang (1995). The static WCET analysis of the implemented F&I barriers in Section 4.3.4 shows that they already provide promising worst-case performance.

Large data-parallel programs from the high-performance domain, which use loop parallelisation for instance, are mostly implemented using either decentralised or hierarchical barriers, mostly with hardware support. Decentralised barriers, often also called broadcast barriers (see Xu et al. 1992), compute termination of a barrier iteration only on local data, and synchronise globally on broadcast messages.

Hierarchical barriers are mostly implemented using a tree-based hardware or software structure. Research on hierarchical barrier implementations is mainly targeting increased average-case performance and average-case efficiency for highly contended resources. Some prominent examples are *Butterfly* barriers introduced by Brooks (1986), and, based on those, *tournament* barriers proposed by Hensgen et al. (1988). Butterfly barriers provide a distributed structure without critical sections, but requiring more memory space than barriers with an accumulating counter. They perform well for a high amount of processes, especially if they are to the power of two, as they synchronise pair-wise. Butterfly barriers are also not prone to the reinitialisation problem. For the *tournament* barriers, Hensgen et al. (1988) introduce double buffering and sense switching to resolve the reinitialisation problem. A similar approach has been used for the subbarriers, which are analysed in this thesis. Also, the implemented F&I barriers are not prone to the reinitialisation problem (cf. Section 3.3.6).

Ramakrishnan and Scherson (1999) present *multiple disjoint barrier synchronizations* (MDBS) to overcome the partitioning problem of nesting in tree-based hardware barriers. The target of their techniques are highly data-parallel programs, which cannot be executed efficiently with just software barriers, e.g. on the *Thinking Machines* CM-5[8].

Sartori and Kumar (2010) propose hybrid hardware/software barrier implementations for chip multiprocessorss (CMPs). They aim to provide support for adaptive, fine-grained barrier implementations, without the need of overly expensive hardware barriers in many-core processors.

Other software barrier implementations[9] for shared-memory multiprocessors using the F&A respectively the CAS primitive are presented and evaluated by Mellor-Crummey and Scott (1991b) (among other software synchronisation techniques applying the F&A primitive).

[8]The CM-5 with 1024 cores was number one in the Top 500 Supercomputers (`http://www.top500.org/list/1993/06/100` in June 1993. [last accessed: April 2013]

[9]The website `http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html` presents further software barrier implementations including pseudo-code. [last accessed: April 2013]

Non-Blocking Synchronisation Techniques

Several publications cover the use of *non-blocking* synchronisation techniques instead of conventional lock-based synchronisations: Anderson et al. (1997), Dechev and Stroustrup (2009), Fraser and Harris (2007), Gao and Hesselink (2007), Ha-Hoai and Tsigas (2003), Herlihy (1988, 1991a, 1993), Herlihy et al. (2003), Michael (2004), Valois (1995).

Non-blocking implementations introduce the advantage of allowing progress in the presence of faulty processes. In lock-based implementations with critical sections, in general, a process being faulty while accessing shared data in a critical section would prohibit progress for all processes needing to access that critical section. For non-blocking concurrent data access, a faulty process either only prevents its own progress, but not the progress of other processes (*lock-free* property), or, with the help of completion checks of other processes, faulty processes are detected and can be restarted (*wait-free* property) (see Fraser and Harris 2007, Herlihy 1988, 1991a, 1993, for details). Non-blocking techniques do not need to mutual exclude other processes so that they could be seen as optimistic concurrency control, as, on the other hand, lock-based techniques are pessimistic. However, optimistic concurrency control is yet still difficult for static timing analysis of parallel HRT programs; non-blocking techniques for parallel HRT programs are yet still to be researched in depth (see also transactional memory below).

Another advantage of non-blocking techniques is that they prevent deadlocks/livelocks (cf. Hennessy and Patterson 2003, Ungerer 1997, p. 559, respectively p. 56), as through optimistic concurrency control it is ensured that processes can not lock each other, that is the other processes would retry if their accesses fail (see Valois 1995). So, non-blocking synchronisation is not prone to priority inversion (see Anderson et al. 1997, Greenwald and Cheriton 1996), which is a common problem for lock-based synchronisation techniques. Nevertheless, there exist a number of solutions to overcome priority inversion of locked-based techniques in real-time systems, and also multiprocessor real-time systems with multiprogrammed workload (see Chen and Tripathi 1994, Chen et al. 1994, Chen and Lin 1990, 1991, Rajkumar 1990, Rajkumar et al. 1988, Sha et al. 1990, 1991). In parallel HRT programs on multi-core processors, which are the baseline programs in this thesis, all threads of one parallel program share the same priority, so they are not exposed to priority inversion, and are executed on different cores (one HRT thread per core, see Section 3.1.1). Also, the scope of this thesis is on the communication and synchronisation of parallel threads below the scheduling layer; issues on synchronisations, arising from HRT threads with different priorities accessing the same shared memory location (or being executed on the same core), would need to be handled on a higher abstraction layer, e.g. with the techniques from the publications cited above. However, the MERASA multi-core processor allows for executing HRT threads in concert with NHRT threads through the use of SMT-cores. On the core-level, a hardware scheduler prioritises HRT threads over NHRT threads to enforce isolation (see Mische et al. 2010). The impact arising from mixed-criticality execution on shared resources, that is e.g. the global memory, is included in the static timing analysis in this thesis by computing WCMLs (see Section 4.2). A similar solution could be used to also support more than one parallel HRT task being executed and analysed on a multi-core processor.

The prerequisites are then that only one HRT thread of a multithreaded task, respectively one singlethreaded HRT task, is executed per core (isolation), and that the impact of HRT tasks on shared resources can be statically computed (as shown in Section 4.2).

A major impact on atomic hardware primitives implemented in today's hardware architectures has sparked from theoretical observations by Maurice Herlihy. Herlihy (1988, 1991a) has proven that some atomic hardware primitives—like atomic registers, TAS, and F&A—are not sufficient for wait-free implementations (for more than two concurrent processes). Herlihy (1988, 1991a) proposes using the CAS primitive that is sufficient for universal wait-free implementations, and hence has been implemented in many hardware architectures and ISAs. Later, Herlihy et al. (2003) introduced an extenuated property called obstruction-freedom, which Herlihy claims to be of more practical use than the wait-free property, because of the overly complex implementations and only theoretical benefit of wait-free algorithms. Obstruction-free data structures should exhibit the same properties as lock-free variants, however, having the advantage of being simpler and hence more efficient. Though, the below cited, recent publications mostly only cover the lock-free property so far.

Massalin and Pu (1992) present an OS for multiprocessors which is based only on lock-free synchronisations. They implement a *two-word CAS* to atomically update a pointer and the location it is pointing to for efficient access in simple data structures like stacks, queues, and linked lists. Shared data, which is too large for being changed with a two-word CAS, is accessed by specifically designed shared objects, that is LIFO stacks and FIFO queues. For processors which do not support a two-word CAS operation, they provide an emulation of the two-word CAS behaviour through kernel-level operations.

Herlihy (1993) presents implementations for lock-free concurrent data structures, that is the access to shared objects is not secured by a critical section, but using the LL/SC primitive to conditionally attempt to access shared data. The lock-free property requires that after a bounded number of attempts, the access is successful. The presented techniques are intended for fault tolerance, and do not focus on aspects indispensable in real-time systems. Hence, it is difficult to obtain a (not overly pessimistic) upper bound on the number of needed attempts for non-blocking data accesses, that are e.g. conditional-stores, to be successful. Furthermore, the proposed techniques for non-blocking access still exhibit much lower performance than simple TAS-based spin locks.

An efficient implementation of lock-free linked lists as building blocks for further data structures is presented by Valois (1995). Valois (1995) only uses the CAS primitive and claims similar performance of the presented lock-free implementation to typical lock-based ones. He also points out that for linked lists, the major problem of the lock-free implementation is the problem of concurrently deleting elements from a linked list, while insert and traversal operations are more or less straightforward. That is for a delete operation, not only the node in the linked list must be removed, but also its next pointer must be invalidated to disable concurrent updates from other processes on this node (cf. Fraser and Harris 2007). Also, the *ABA problem* of CAS primitives is described, and operations to overcome it are introduced. Together, the delete operation gets rather complicated, including auxiliary nodes to prevent data inconsistencies. One key aspect, for the scope of real-time systems, learned from the paper is that a single operation of a lock-free list is impossible to be bounded, however, it can be bounded for a number of accesses. Still, it shows how difficult it is—depending on the implementation even impossible—to compute safe upper bounds for single accesses to lock-free data structures. Despite claiming that future results could show that the presented lock-free implementation is competitive to lock-based approaches, no such results have been published yet.

Anderson et al. (1997) evaluated scheduability of real-time programs with lock-free access for interprocess communication, called object sharing, on uni-processors. They show that specific scheduling restrictions could lead to improved performance of lock-free accesses in comparison to lock-based accesses, especially if lock-based implementations need to adhere to protocols to prevent priority inversion. The claimed HRT program analysed is a video conferencing system, which is more likely to have SRT requirements than HRT ones. The results originate from an uni-processor program, running multiple threads, and thus cannot be easily adapted to multi-core processors, especially since interference is bounded by only one thread being processed at a time. Retries of optimistic concurrent accesses increase, if more threads are executed in concert, e.g. as typically on multi-cores. It might then be complex or infeasible to conduct a static timing analysis.

Michael (2004) introduces so-called *hazard pointers* for dynamic reuse of memory nodes for lock-free objects. It is related to the reference counting introduced by Valois (1995), however, *hazard pointers* exhibit better average-case performance on up to four processors, especially if the number of threads per processor is increased. Also, the lock-free approach with *hazard pointers* shows better average-case performance than lock-based synchronisation through dynamic memory reclamation. Though, both approaches still suffer from possible memory space overhead. Furthermore, for real-time systems, coding guidelines mostly prohibit or restrict the use of dynamic memory management; the worst-case performance in real-time systems is difficult to predict then, as is the usability of *hazard pointers* in real-time systems.

Proving that lock-free algorithms are correct, is a difficult and error prone task. Gao and Hesselink (2007) present a reduction theorem to a lock-free pattern applying the CAS primitive variation based on Herlihy's LL/SC approach (see Herlihy 1991b, 1993). By that they claim to reduce the effort to formally prove correctness of lock-free algorithms, which they state to be worthwhile.

Dechev and Stroustrup (2009) present a study of non-blocking synchronisation with CAS respectively software transactional memory (STM) for concurrent synchronisation in embedded real-time software for future robotic spacecraft. Their implementation of a lock-free shared vector shall achieve higher safety and a performance gain on a COTS multi-core processor for SRT tasks.

Engdahl and Chung (2010) present lock-free data structures for real-time control programs on an ARM COTS multi-core processor. They claim that lock-free access to data structures fits better to the asynchronous behaviour of control applications as traditional lock-based algorithm do. However, the work-in-progress is missing a formal prove of their proposed implementations yet. Also, the authors note that the runtime behaviour is not deterministic due to retries in their enqueue and dequeue implementation, which could then disqualify them for HRT systems, if a (static) timing analysis is not possible.

Transactional Memory

Inspired from non-blocking algorithms and to relieve the difficulties of concurrent programming, *transactional memory* has been in the focus of research by a number of authors: for example Herlihy and Moss (1993) and Fraser and Harris (2007). A survey on transactional memory systems, that is STM and hardware transactional memory (HTM), is presented by Harris et al. (2010), Larus and Rajwar (2007), but without specifically presenting aspects of transactional memory in real-time systems. An approach on STM for multiprocessors, in the domain of general purpose computing, is presented by Saha et al. (2006), also without specifically targeting real-time systems.

Lately, various authors tackled the use of transactional memory (STM and HTM) for real-time systems: Manson et al. (2005), Meawad et al. (2011), Schoeberl et al. (2010), Schoeberl and Hilber (2010), Sarni et al. (2009), Fahmy et al. (2009), Maldonado et al. (2011), and Barros and Pinho (2011).

One of the first publication targeting transactional memory in real-time systems has been presented by Manson et al. (2005). They introduce a restricted STM system, called Preemtable Atomic Regions (PAR), for multitasking programs in distributed embedded systems running real-time Java (see Bollella et al. 2000) on uni-processors. The target of PAR is to reduce worst-case response times (WCRTs) of high priority tasks. PAR guarantees that a sequence of instructions is executed atomically, whereas actions of a lower priority task can be undone if a higher priority task is released. The interrupted task is then re-executed by PAR afterwards. The approach is semantically similar to priority ceiling protocols (e.g. Chen and Lin 1990, Sha et al. 1990, 1991) when using a lock-based approach for critical sections, but, as the authors claim, achieving faster execution times and better scalability for distributed, real-time embedded Java systems.

Further work on Java-based real-time systems, but with HTM support in Java CMPs, is presented by Meawad et al. (2011), Schoeberl et al. (2010), Schoeberl and Hilber (2010). Schoeberl et al. (2010) and Schoeberl and Hilber (2010) propose a real-time transactional memory (RTTM) as an extension to a real-time capable Java CMP. The authors claim that RTTM provides upper bounds on maximum transaction retries for periodic threads, while also allowing for more simple programming of concurrent programs than traditional lock-based programming does. The cited publications focus on wait-free implementation of queues using the RTTM approach versus CAS-like approaches. The authors also state, that a wait-free bounded capacity queue, which usually needs CASN (cf. Ha-Hoai and Tsigas 2003, Harris et al. 2002) (also called MCAS) support, can be implemented with RTTM using micro-transactions (see Meawad et al. 2011). RTTM uses late conflict detection in the commit phase, and is only applied on memory operations in short atomic section. Schoeberl and Hilber (2010) also present a possible timing analysis when applying the RTTM. In detail, they restrict the number of transactions per period to one per thread, however, the authors claim that the analysis is also safe for more transactions in one period, but overly conservative so far; the aim of their future work is to tighten these bounds.

Sarni et al. (2009) present the RT-STM as the first STM implementation for real-time scheduling of transactions in SRT programs on multi-core processors.

They implement a transaction scheduler considering deadlines for transactions, that is the deadline is incorporated by the scheduler to decide on either a transaction is aborted, or supported to complete. Sarni et al. (2009) state that their RT-STM implementation is an enhancement of a previous STM proposed by Fraser (2004). Sarni et al. (2009) employ for their time measurements a slightly changed red-black tree benchmark (see Fraser 2004), and LITMUS^{RT} (LInux Testbed for MUltiprocessor Scheduling in Real-Time systems) (cf. Brandenburg et al. 2008). The RT-STM approach outperforms other STM approaches under various scheduling policies in terms of keeping SRT deadlines. However, only around 56 % of deadlines are guaranteed to be hold with RT-STM for the red-black tree benchmark, which might be sufficient as they aim for SRT systems, but not for HRT systems.

Fahmy et al. (2008) motivate the use of STM in distributed real-time embedded systems. They especially emphasise the problem of lock-based programming to provide deadlock-freedom, e.g. in the case of failures. Fahmy et al. (2009) present a framework that incorporates transactions that consist of so called sub-transactions on every processing element (node). The sub-transactions are then subject to crash failures on the corresponding node. The authors especially focus on WCRT analyses of periodic tasks (sub-transactions) taking into account remote procedure calls (RPCs) in a distributed system.

Maldonado et al. (2011) present scheduling mechanisms for reactive programs with STM, which adaptively adjust the execution mode depending on a task's laxity and annotated duration length of its transactions. For that, Maldonado et al. (2011) estimate the duration of transactions and switch between different transaction handling modes, that is more optimistic in the beginning, and, if the deadline is near (low laxity), they switch to more deterministic and pessimistic transaction modes. To achieve such a behaviour, an existing STM (TinySTM library, see Felber et al. 2008) has been enhanced, and the Linux OS scheduler has been augmented to allow for prohibiting preemption and migration for transactions which are close to their deadline.

Based on the above mentioned approaches, Barros and Pinho (2011) state that these publications do not fully capture and cover all aspects for contention management of transaction management of transactional memories for parallel embedded real-time systems. Therefore, they propose first insights on an STM-based approach incorporating on-line information. Barros and Pinho (2011) claim that their approach can thereby reduce the overall number of retries and increase the responsiveness of tasks. Also they state that their approach reduces the wastage of computing power on aborted transactions and guarantees that deadlines are kept. The authors employ a multi-versioned STM, for which the number of version for each object are computed off-line. Then, the authors argue that read-only transactions never conflict with other concurrent transactions, as they only rely on consistent snapshots of their prior read-sets (cf. Cachopo and Rito-Silva 2006). Hence, they state that contention must only be covered for update transactions. Preliminary, Barros and Pinho (2011) propose a pessimistic solution by ordering conflicting transactions chronological by an additional algorithm executed in parallel at each transactional update commit. Beside the theoretical argumentations, no comparing evaluation results are provided so far by Barros and Pinho (2011).

The strongest of all warriors are these two—Time and Patience. – Leo Tolstoy

4 WCET Analysis of Synchronisations

In this chapter the details on WCET analyses of the proposed software synchronisation techniques from Chapter 3 are shown. First, in Section 4.1 an introduction on the principles of WCET analyses for parallel programs are given. In Section 4.1.3 the settings for the static WCET analyses with OTAWA are introduced, and followed in Section 4.2 by the mathematical analysis of WCMLs for memory operations in a bus-based, shared-memory multi-core processor with the augmented memory controller and RMW operations (see Section 3.2). The WCMLs are then used in the static timing analyses and WCET comparisons of the software synchronisations introduced in Section 3.3. The analysed synchronisation techniques are further evaluated in the static WCET analysis of two parallel programs in Section 4.4. Related work of WCET analyses of software synchronisation techniques and parallel HRT programs is presented in Section 4.5.

4.1. Introduction on WCET Analyses of Parallel Programs

The WCET analyses of parallel programs is not too different from WCET analyses of sequential programs that already share resources, e.g. I/O devices, in distributed embedded systems. However, additional difficulties are introduced from synchronisation of memory accesses in parallel programs, and the tight coupling in shared-memory multi-core processors. The main issue for a static timing analysis of parallel programs is to find a safe upper bound for concurrent threads that use synchronisations to access shared resources, and how to keep possible overestimation and pessimism as low as possible. This is especially difficult, as the analysis has to be done with unknown information on competing accesses to shared resources, that is one cannot be always sure at which execution point HRT threads on the cores of a multi-core currently are. However, there are points in a parallel program, e.g. synchronisation points at barriers, that can be used in the static WCET analysis to synchronise the state of HRT threads on all cores.

4.1.1. Timing Analysability and Timing Predictability

For WCET analyses, two key terms can be distinguished, that is *timing analysability* and *timing predictability*: a system, program, or function is *timing analysable*, if it is possible to compute a *safe* upper bound, that is an estimated WCET that is greater or equal to the *unknown* WCET. *Timing predictability* extends *timing analysability* by a qualitative measure of how good a computed upper bound (estimated WCET) is, or, in other words, as defined by Thiele and Wilhelm (2004), as the pessimism in the WCET respectively in the BCET. Often this is also described as how *tight* an estimated WCET is, that is how close the estimated WCET is to the real but unknown WCET (see also Section 4.5). The major drawback of *timing predictability* or WCET *tightness* is that it is difficult to quantify, as the real WCET is not known. However, *tightness* is, besides the worst-case performance, an important quality factor in HRT systems.

4.1.2. Pessimism and Overestimation

The variations in the WCETs that degrade the *timing predictability* in the WCET analysis can be categorised into *WCET overestimation* and *WCET pessimism.* Roughly said, *pessimism* is introduced from unknown, but realistic behaviour, whereas *overestimation* stems from unknown and (most likely) unrealistic behaviour or assumptions in the WCET analysis which are needed to safely upper bound the execution times.

In detail, pessimism in the WCET analysis of parallel programs originates from unknown but realistic behaviour of concurrently executed threads. For instance, if in a multi-core processor several threads access shared resources, like a shared global memory, in the WCET analysis this interference and competition on the shared memory and the memory interconnect has to be taken into account. Therefore, the WCML of one thread is based on the access patterns from other threads. Now, if there are memory accesses that have different access times (e.g. RMW operations and usual load/store operations), the longest access time has to be assumed for the other threads in the analysis of the WCML of one thread. Even if not all of those accesses to the shared memory are RMW operations, the WCET analysis has to assume so, and thus the pessimism caused by this assumption has to be included. It is mostly not possible to overcome all possible pessimism, however, in Chapter 5 the *Split-Phase Synchronisation Technique* is introduced that aims at reducing the pessimism in the static WCET analysis triggered from concurrent accesses with differing memory latencies. Additionally to pessimism, WCET *overestimation* is introduced from unknown, but (mostly) unrealistic assumptions. Overestimation is either triggered if not all information, e.g. on input or user data, are known at design time for a static timing analysis, or if the analysis would get to complex to include all possible information and paths and hence being rendered infeasible. An example of WCET overestimation could be the interference that is based on threads or interrupt service routines (ISRs) that are executed only on specific external conditions, events, or on specific input data or user input at runtime. Thus, in some cases overestimating the WCET must be accepted, to ensure a safe upper bound.

Pessimism and especially overestimation in parallel HRT programs can be reduced—with the help of the programmer—by following specific coding guidelines (see for example Bonenfant et al. 2010, Ozaktas et al. 2013). Coding guidelines were also developed in the EU-project MERASA[1]. Also, coding standards, like *MISRA-C*[2], influenced the development of above mentioned coding guidelines, and might also be taken into account. However, already for sequential programs, the use of coding standards alone, does not solve all issues for static timing analyses (see Gebhard et al. 2011). Therefore, an approach to limit possible pessimism and overestimation, and ease static timing analyses of parallel programs is pursued by applying *parallel design patterns* and *synchronisation idioms*, as part of the EU-project parMERASA[3], shortly introduced in Chapter 6.

[1]See `http://www.merasa.org` for coding guidelines developed in the EU-project MERASA.

[2]Specific coding standards for the C and C++ programming language, e.g. for the automotive domain, are available from the MISRA group for purchase at `http://www.misra.org.uk/`

[3]See `http://www.parmerasa.eu` for coding guidelines and the parallelisation approach for predictable execution of HRT programs to be developed in the EU-project parMERASA.

4.1.3. Static WCET Analysis of Parallel Programs

Hardware architectures used for time-critical tasks in HRT systems were less complex in the beginning than they are today. As stated by Puschner and Schoeberl (2008), architectures used in HRT systems became more and more elaborated since the early 90's, hence creating a need for sophisticated timing analysis techniques. By then, analysis of timing behaviour became a major research subject (see Puschner and Burns 2000, Rochange 2011, Wilhelm et al. 2008). Nowadays, the need for higher performance and better power efficiency is a driving force of spreading multi-cores into the domain of embedded computing, thus putting additional pressure on analysing the timing behaviour of critical tasks to cope with multi-core architectures. Rochange (2011) states that techniques to manage scheduling for real-time tasks in multi-processor and multi-core environments has drawn much attention in the recent years, mostly assuming the problem of analysing parallel architectures and parallel programs—essentially interference between parallel tasks and threads—has been solved. The main point in this observation stems from the fact that, until recent, timing analysis techniques assumed isolation of the analysed task. Hence, the analysis of parallel programs, which require some sort of resource sharing, e.g. synchronising on shared data or competing accesses to shared resources, needs to take those interferences and interactions into account. Rochange (2011) clarifies that this does not only hold for static analysis techniques, but for measurement-based analysis techniques as well: it is highly unlikely that all possible paths in a parallel program running on a parallel architecture are to be observed.

Static WCET analysis techniques target to determine a safe upper bound on the real, but unknown WCET (see Section 2.1.1). Figure 4.1 shows the typical computation and analysis flow for a static WCET analysis in general (cf. Wilhelm et al. 2009c). Wilhelm et al. (2009c) and Rochange (2011) state that it can be categorised into three phases: (1) building the control flow graph (CFG) from a flow analysis of the binary executable code and flow facts, such as loop bounds and infeasible paths from the source code representation and annotations, (2) low-level (microarchitectural) analysis computes the bounds of basic blocks, and (3) finding out the longest path and its execution costs in the global bound analysis of the whole program (the three major phases are the differently coloured (blue, green, and red), rectangle-shaped elements in Figure 4.1).

In this thesis, the open-source static WCET analysis tool OTAWA is used. It implements state-of-the-art algorithms for WCET analysis (see Ballabriga et al. 2010). It supports the used target multi-core processor, the TriCore-based MERASA ISA, and accounts for possible contentions on the shared bus and memory controller by considering WCMLs. However, in the case of possible timing anomalies (see Reineke and Sen 2009), it is not safe to only account for WCMLs, as then all possible memory latencies need to be considered (cf. Rochange 2011). The microarchitectural analysis step includes the analysis of the local memories of the MERASA processor, that is the DSP and the D-ISP. In the following analyses it is assumed that no replacement in the D-ISP takes place. Thus, once a function has been loaded into the scratchpad, it is persistent until the end of the analysis; however, Metzlaff (2012) thoroughly analysed the effects of different replacement strategies of the D-ISP.

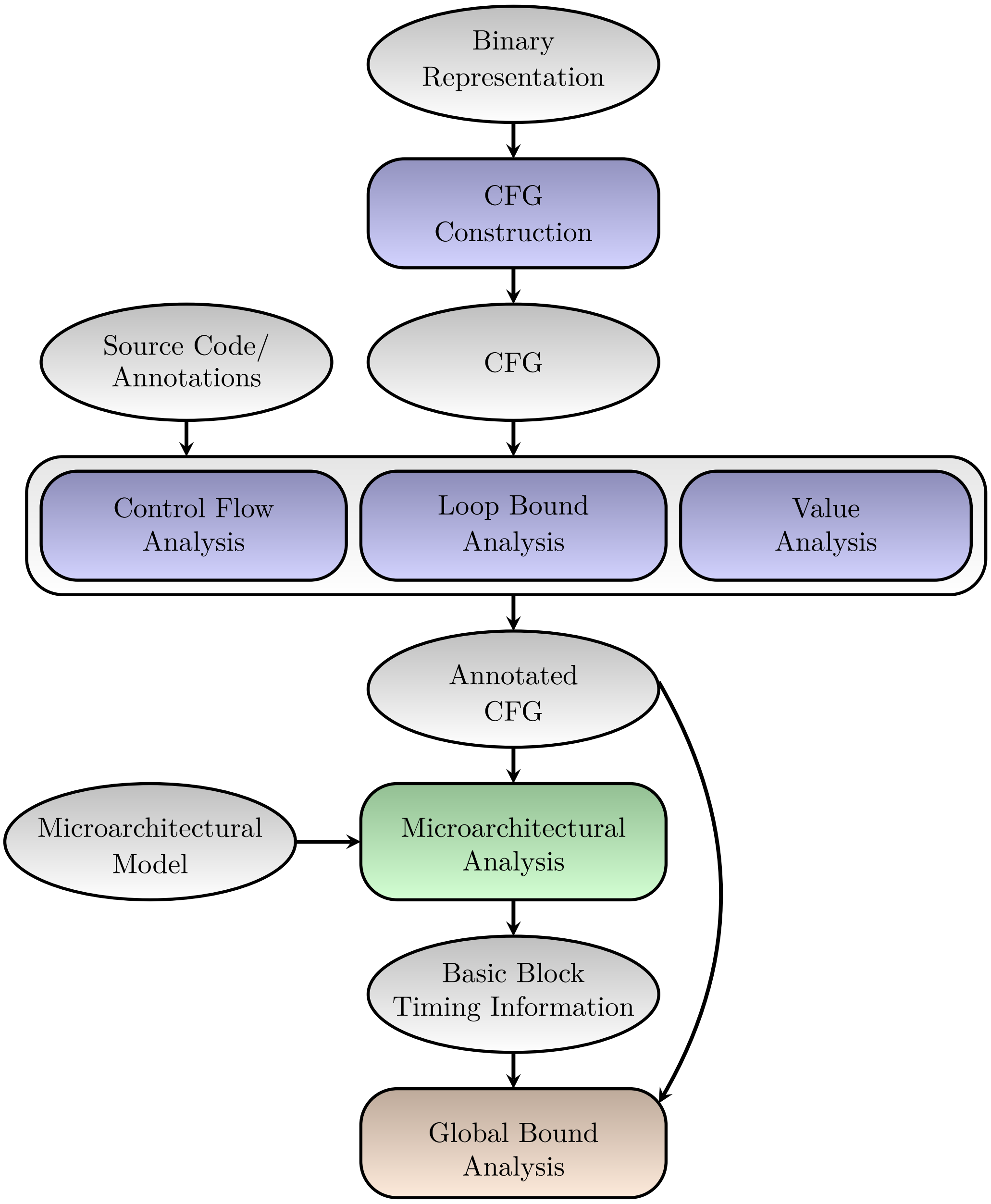


FIGURE 4.1.: Typical computational flow of static WCET analysis (cf. Figure 1 from Wilhelm et al. 2009c). The different coloured rectangles represent computational phases, which can be categorised into three major phases, and elliptic shaped elements illustrate data representations, e.g. the *Binary Representation* as starting point, the constructed CFG, annotations, and the underlying *microarchitectural model*.

The static WCET analysis with OTAWA for parallel programs is similar to the typical static WCET analyses of sequential programs. The main difference and also challenge is that interferences, which induce waiting times for parallel executed threads, need to be taken into account. In the following analyses, interferences are limited to the global memory and memory interconnect, that is threads are isolated inside the SMT-cores, and only the memory interconnect is accessed concurrently from all cores. Similar actions in the analyses need to be taken into consideration for further shared resources, e.g. I/O devices. However, techniques introduced in this thesis in the shared memory controller would not change, but the arbitration techniques in the interconnect might be different, depending on the interconnect architecture, e.g. single buses, multi buses, or even Network-on-Chips (NoCs). The access to the shared memory is scheduled through an arbiter in the real-time bus interconnect (see Paolieri et al. 2009a, Ungerer et al. 2010), and WCMLs (see Section 4.2) are integrated in the WCET analyses (see Gerdes et al. 2012a,b, Rochange et al. 2010, Wolf et al. 2010a), as e.g. also proposed by Puschner and Schoeberl (2008) for CMPs. Also, the WCET analyses in this thesis use the concept of separating execution and waiting times to analyse synchronisations as introduced by Rochange et al. (2010), Wolf et al. (2010a, 2011) and Gerdes et al. (2012b). They propose to split the worst-case waiting times (WCWTs) into three subcategories:

- waiting times that are analysed *independent* of the number of concurrently running threads and the program context,
- waiting times that are analysed *depending* on the number of concurrently running threads, but *not depending* on the program context, and
- waiting times that are analysed *depending* on the number of concurrently running threads, *and* the program context.

Section 4.3 presents static WCET analyses including WCWTs in detail for the software synchronisation techniques proposed in Chapter 3.

Another widely spread approach of WCET analyses is based on measurements on the real hardware. An example of such a measurement-based analysis with the commercial WCET tool *RapiTime* (see Rapita Systems Ltd. 2011) on the MERASA multi-core processor is described shortly in a case study on parallelising and analysing the control software of a large drilling machine by Gerdes et al. (2011). As stated by Wilhelm et al. (2009c), one drawback of measurement-based timing analyses over static analyses is the problem of exactly measure execution times for complex processor architectures, e.g. multi-cores with (shared) caches, or complex memory hierarchies. The main problem is then not only the code coverage, but also the coverage of all possible hardware states, especially of all possible initial states (see Wilhelm et al. 2009c). Measurement-based and static approaches to timing analyses both have to suffer with large variabilities in the execution time from complex hardware features like branch predictions and caches (see Colin and Petters 2003). Especially effects from caches highly influence the precision of WCET estimates as shown by Wilhelm et al. (2009c). Petters et al. (2007) propose a combination of measurement-based and static WCET analysis. The main problems of measurement-based approaches are induced by possible missing context and flow information, as stated by Petters et al. (2007).

In the MERASA[1] project a combination of measurement-based and static WCET analyses has been chosen[4]. The additional flow facts derived from measurements, which are difficult or even not possible to be derived from the source/binary code, are included in the static WCET analysis to produce tighter WCET estimates (see Ungerer et al. 2010). Missing and insufficient annotations from coding guidelines[2] could also be balanced with this combined approach. For the parMERASA[3] project annotations for the static timing analysis are derived from applying parallel design patterns (see Chapter 6).

Evaluation Settings for Static WCET Analyses

The memory latencies used in this thesis have been derived from the MERASA FPGA prototype. In the WCET model of the MERASA multi-core processor, the bus cycle time is fixed at 1 cycle. The time a load takes in the memory controller is devised to take 5 cycles, whereas a store takes 4 cycles. A store operation is handled faster than a load operation, as no actual return value needs to be transferred back to the core. However, a notification that the store was successfully finished is returned over the bus to the core, so the store operation will not spare the bus cycle time after the memory controller finishes the memory operation. The RMW operations—that is the TAS and the F&I/F&D operations, consist of a load, a modification, and a store—take more time. For a TAS operation, no actual manipulation needs to be done, therefore a TAS operation just needs to load a value, and then store back a constant value (e.g. ’0’ or ’1’). Hence, a TAS operation takes 9 cycles, that is the sum of the 5 cycles (load) and 4 cycles (store). For a F&I respectively a F&D operation, the loaded value needs to be incremented or decremented. Thus, an additional cycle is needed to manipulate the loaded value before it is stored back. So, the time of a F&I/F&D operation sums up to 10 cycles, that is 5 cycles (load), 1 cycle for the increment/decrement, and 4 cycles (store). For other multi-core processors, the memory latencies are different, depending on the implementation and memory controller. However, the memory latencies could then be substituted to compute the WCMLs for further multi-core processors and memory controllers.

In the following, the WCMLs are given as the upper bound delay on a HRT memory request from when it is ready to be dispatched to the shared memory (over the bus), until it is successfully finished and a following request could be dispatched. The memory requests from all cores to the global shared memory are arbitrated by a real-time aware bus (see Figure 3.1) in the MERASA processor. The bus arbitrates accesses in a round-robin fashion between cores, that is by time-division multiple access (TDMA). When a memory request from a core is accepted and dispatched to the bus and subsequently to the memory controller, follow-up memory requests, from the same core, are dispatched after the previous access has been finished (cf. Section 4.2.1). The bus is treated as *full duplex*, meaning that a request from a core to the memory controller and a result from the memory controller to a core can be dispatched over the bus at the same time.

Further remarks and details on special restrictions in the evaluation settings are pointed out in the corresponding section, e.g. the costs of calls and returns for the analysis of software synchronisations in Section 4.3.

[4]see public Deliverable D3.6 and D3.7 “WCET tools for the HW demonstrator” at www.merasa.org

4.2. Worst-Case Memory Latencies

The WCETs of parallelised HRT programs running on shared-memory multi-core processors are highly depending on the knowledge of competing accesses to the shared memory and the WCMLs. The latency for a memory request is split into three parts: 1) the time the bus needs to dispatch the memory request from a core to the memory controller, the so-called bus cycle time, 2) the time the memory controller needs to execute the memory request, which is depending on which kind of memory request is executed, either a load, a store, a TAS, or a F&I/F&D operation, and 3) again the bus cycle time to return a value to the core that requested the memory operation.

Figure 4.2 depicts the worst-case memory latency of a HRT memory request of one core, namely *Core 1*, for a MERASA quad-core processor. The HRT memory request could be either a normal load/store operation, or a RMW operation. Also, the Figure 4.2 shows the overlapping of accesses when different shared resources—the shared memory and the bus interconnect—are used. However, memory operations are served sequentially on a FCFS basis by the augmented memory controller (see Section 3.1.2).

To determine the WCMLs of different HRT memory requests, namely a normal load, a normal store, or a RMW operation (TAS, F&I, or F&D), two situations need to be covered: on the one hand, as SMT-cores are employed (see Section 3.1.1), a HRT memory request might be delayed by a NHRT memory request on the same core that was, in the worst case, dispatched just one cycle before the HRT memory request is ready to be dispatched. On the other hand, the delay introduced from memory request of other cores adds up on the time the HRT memory request of *Core 1* takes. For the first case, the delay introduced from a NHRT request is independent of the number of cores N. But, it has to be assumed that this NHRT memory request is a slow memory request, that is the type of memory request that takes the most time. In the following, this delay will be defined as T_{max}, where $T_{\text{max}} = T_{\text{Load}} + T_{\text{Modification}} + T_{\text{Store}}$. As mentioned in Section 4.1.3, a store operation is one cycle faster than a load operation.

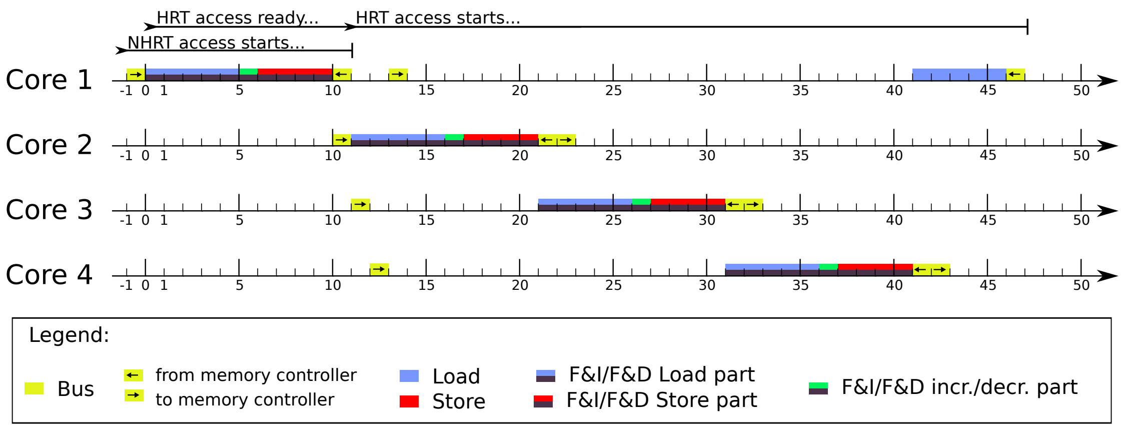


FIGURE 4.2.: Worst-case Memory Latencies for a HRT memory accesses of *Core 1* in a MERASA quad-core processor with five cycles load latency.

Assuming also one cycle as modification latency $T_{\text{Modification}}$ for a RMW operations, it follows that $T_{\max} = 2 \cdot T_{\text{Load}}$. So, in the worst case, a delay of $T_{\max} - 1$, introduced from a NHRT memory request, has to be taken in account for the HRT memory request of *Core 1*. For the latter case, the introduced delay depends on the number of cores. That is for an N-core processor it adds an additional delay of $(N - 1) \cdot T_{\max}$, as in the worst case the memory request of each of the other $N - 1$ cores are handled before the HRT memory request of *Core 1*. Also, the extra bus cycle T_{B} to return a value from the memory controller to the core needs to be taken into account. Finally, the time T_{HRT} that the HRT memory request of *Core 1* needs, must be added. The bus cycle time T_{B} only needs to be taken into account for the NHRT and HRT memory access of *Core 1*, as by employing a *full duplex* bus, the other bus cycle times are hidden (see overlapping of bus accesses (yellow boxes) in cycles 10, 13, etc. in Figure 4.2).
In summary, the WCML T_{WCML} in the N-core MERASA processor adds up to:

$$T_{\text{WCML}} = \overbrace{T_{\text{HRT}} + T_{\text{B}}}^{\text{HRT access}} + \overbrace{T_{\max} + T_{\text{B}}}^{\text{NHRT acceess}} + \underbrace{\overbrace{(N-1) \cdot T_{\max}}}_{\text{Other N-1 cores}} \tag{4.1}$$

Equation 4.1 can be easily combined and re-written as:

$$T_{\text{WCML}} = T_{\text{HRT}} + 2 \cdot T_{\text{B}} + N \cdot T_{\max} \tag{4.2}$$

Including the bus cycle time, it is possible to derive the WCML T_{WCML} with the above depicted Equation 4.2. Figure 4.3 shows the WCMLs of a load operation for four to eight cores, and with a memory latency for a load of five to ten cycles (respectively four to nine cycles for a store). The grey scale (depicted on the right top side of the figure) shows the WCMLs: the darker grey depicts a low WCML, whereas the higher the WCML gets, the lighter the grey is (the numbers are also presented in Table 4.1).

Figure 4.3 shows that doubling up the number of cores or doubling up the load latency has nearly the same effect on the WCMLs (see lower right corner for ten cycles load latency and four cores, and upper left corner for five cycles load latency and eight cores in Figure 4.3 showing the same mid-grey colour, meaning that the WCMLs are in a close range). Both variables, load latency and amount of cores, are in the dominating summand $N \cdot T_{\max}$ of Equation 4.2, which is the baseline for Figure 4.3. However, when the load latency is increased the effect on the WCMLs is slightly higher because of the summand T_{HRT}, which includes the load latency, but not the number of cores. This is discussed in more detail in the following Section 4.2.1.

Please note that the T_{WCML} of a RMW operation, compared to a normal load operation, is only increased by an additional modification cycle and the store latency. Depending on the load latency, this is in the range from five to ten cycles. Also, the WCMLs for load/store operations on a synchronisation variable are the same as for normal loads/stores. However, when the split-phase synchronisation technique is applied, the WCMLs of loads/stores on synchronisation variables differ (see details in Section 5.3.1).

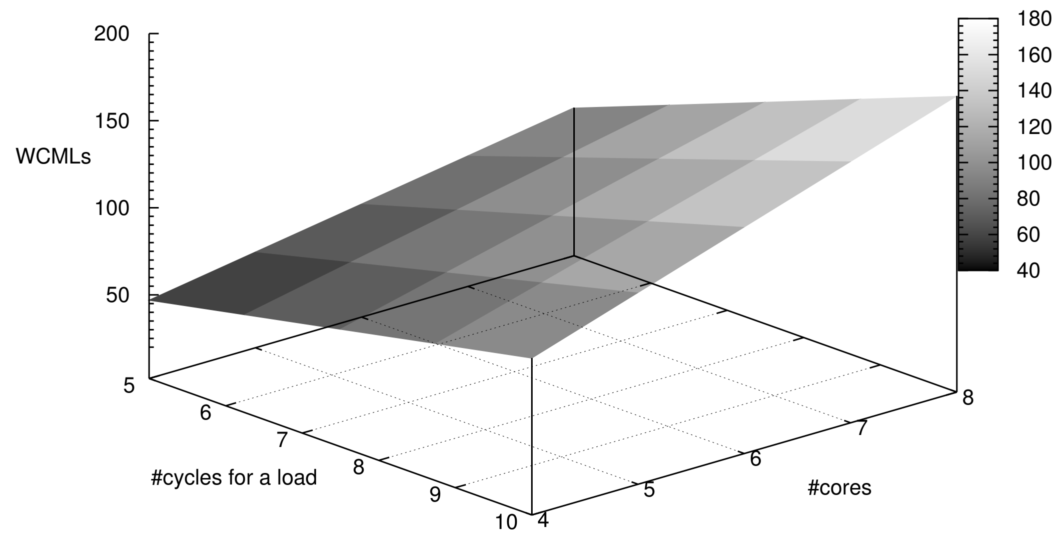


FIGURE 4.3.: WCMLs of load operations in the MERASA multi-core processor with the augmented memory controller depending on the number of cores (4-8) and load latency (5-10 cycles) computed with Equation 4.2.

4.2.1. Effect of WCMLs on the WCET of Parallel Programs

An interesting point about the different configurations is that one cannot make assumptions on which configuration might be better in terms of overall WCET, as well as which configuration has better timing predictability without taking the specific program into account. However, the WCML includes the pessimism on shared resources in the analysis: it is assumed that each core is continuously dispatching a worst-case (in terms of durations) memory operation interfering with the other cores. This assumption safely upper bounds the possible delay introduced on the WCMLs of one core introduced from memory accesses of all the other cores, however, that assumption is not very optimistic and rather unlikely. Then, the timing predictability for eight cores with five cycles load latency could be actually better than for four cores with five cycles latency. This arises from the case that in the eight-core processor it might be more frequently the case that every core dispatches an access to the shared memory, even if it is not a RMW operation all the time. This is the case as the more cores are connected to the bus the higher the contention on this bus gets, and that also leads to the situation where the pessimistic view on WCMLs is becoming more realistic.

Table 4.1.:
Parametric worst-case memory latencies in the MERASA WCET model for four, resp. eight cores, and a memory latency of five, resp. ten cycles (see also Figure 4.3).

# Cores	Memory Latency for a Load Operation			
	5 cycles		10 cycles	
	Memory Operation	WCML	Memory Operation	WCML
4 Cores	load	47	load	92
	store	46	store	91
	TAS	51	TAS	101
	F&I/F&D	52	F&I/F&D	102
	5 cycles		10 cycles	
	Memory Operation	WCML	Memory Operation	WCML
8 Cores	load	87	load	172
	store	86	store	171
	TAS	91	TAS	181
	F&I/F&D	92	F&I/F&D	182

Though, please note that the WCMLs, as computed above, are considered in the static WCET analyses if accesses to the local memories, that is instruction and data scratchpads, are found by the data analysis to be misses. For processors that are free of timing anomalies (see Section 4.1.3), it is, however, safe to upper bound memory accesses with the above computed WCMLs.

The effect of this pessimism on the WCET guarantee also highly relates to the chosen arbitration at the bus interconnect to the shared memory, that is either round-robin arbitration between cores (TDMA), or between accesses (see also Paolieri et al. (2013, 2009a), Ungerer et al. (2010) for the different possible arbitration schemes at the bus interconnect). If a round-robin arbitration between accesses is chosen, it might be possible that some cores get to dispatch more accesses than others, as they use the free slots of other cores. However, this implementation is only useful to speed up the average-case execution time (ACET), as in the static WCET analysis it is not known when a slot is free to be used by another core. Even in the case a thread is e.g. suspended and not issuing any access to the shared memory, it cannot be assumed in the static timing analysis used in this thesis (cf. Section 4.3.4). If a round-robin arbitration between accesses is chosen, it might also be the case that this arbitration strategy violates the strong fairness requirement for synchronisations (critical section requirement) as stated in Sections 2.2.1 and 3.3.

To prevent such a behaviour, the arbitration to the shared memory is done round-robin between cores in a TDMA fashion, and a free slot, e.g. a core that currently has no memory request in its TDMA cycle, would be just unused (see Section 3.2.1). In the static WCET analysis of one core, no free slots are covered, but the pessimistic view that every competing core issues the worst-case memory operation (a RMW operation) every time, as far as no other assumptions on the competing accesses can be safely made.

The duration of one slot in the TDMA arbitration schedule for the bus is chosen to just fit in one memory operation of each core, that is a slot size of one cycle. The effect and benefit of higher bus cycles in the TDMA arbitration for HRT programs depends on the program execution. For instance, Wandeler and Thiele (2006) present an analytical method to determine optimal cycle lengths in a TDMA arbitration scheme in a multiprocessor environment. If changes in the arbitration scheme are conducted, the WCET model and analyses must be aware of those cycle lengths and arbitration schemes to include the effects on the WCMLs in the static timing analysis.

Table 4.1 depicts that the WCML of a load or store is nearly the same as for a RMW operation–TAS or F&I/F&D. Although load operations are more often triggered (e.g. for every instruction fetch that is not in the local memory) than RMW operation (which are only needed for atomic access to (synchronisation) variables), the difference for the WCMLs does not reflect those occurrences. A solution that takes account for those occurrences by prioritising load operations over RMW operations in the augmented memory controller to reduce pessimism in the WCET analysis—the *split-phase synchronisation technique*—is presented and evaluated in detail in Chapter 5.

Another interesting observation can be made from the results in Table 4.1: doubling the memory latency (from 5 to 10 cycles) leads to a doubling of the WCMLs (e.g. for a load operation); however, doubling the number of cores does only increase the WCMLs for a factor around 1.85. That is due to Equation 4.2, in which the memory latency itself occurs two times as a summand, and the number of cores only in one summand. Hence, an increased memory latency has a higher negative influence on the WCMLs than a higher amount of cores. Please note that this is the case as SMT-cores are employed in the MERASA processor, thus the factor of N cores instead of $N-1$ cores in Equation 4.2. If just one thread per core is executed, that is a thread cannot be influenced by other (NHRT) threads of the same core, the impact on the WCMLs would be different: then, the WCMLs increases by a factor slightly bigger than 2 for doubling the number of cores, whereas the factor due to doubling the memory latencies is slightly smaller than 2.

Table 4.1 also shows the baseline configuration for the following analyses and evaluations of software synchronisations and parallel programs in Sections 4.3 and 4.4. That is the different WCMLs (T_{WCML}) for a load, a store, and the two implemented RMW operations of a core in the quad-core MERASA processor with five cycles memory latency (top left of Table 4.1). In Chapter 5 these results are then extended with WCMLs when the split-phase synchronisation technique is applied.

4.3. WCET Analysis of Software Synchronisation Techniques

In the following, the implemented software synchronisations are analysed with the static WCET analysis tool OTAWA. The synchronisation methods are divided into three subsections, that is for software synchronisations using: busy-wait algorithms in Section 4.3.1, blocking algorithms in Section 4.3.2, and software barrier implementations in Section 4.3.3. Section 4.3.4 discusses and compares the WCET estimates of all examined software synchronisations. Results of the static WCET analysis of parallelised HRT programs using those software synchronisations are presented in Section 4.4 and Section 5.3.3 (with the split-phase synchronisation technique). The estimated WCETs depicted in Section 4.3.4 are independent of the possible program context, however, the waiting times can be related to a possible program execution. Therefore, the difference on the impact of different software synchronisation techniques becomes more evident in the full static WCET analyses of parallel programs using those software techniques as shown in Section 4.4.

Evaluation Settings The WCET analyses are done as detailed in Section 4.1.3, using the computed WCMLs from Section 4.2, where applicable. However, some further evaluation settings are introduced from the following restrictions. Calls and returns are expensive in the MERASA prototype (around 80 cycles for each call/return), that is no special technique is used to speed up context switches as it is mostly done in actual hardware architectures. In the analysis of software synchronisations these additional costs are omitted, only the costs for filling the D-ISP need to be taken into account when the first call to a function is a miss. Once in the D-ISP, the function will be persistent and will always hit in the D-ISP. For the analyses of software synchronisations in Section 4.3.4, it is assumed that the analysed synchronisation functions have been already loaded to the D-ISP, and therefore *always hit* in the D-ISP. Also, function calls and returns are mostly omitted from the static timing analysis of software synchronisations (as far as possible), as their overhead in the MERASA architecture is high, and could have been reduced by implementing techniques that speed up context switches (as in embedded COTS processors[5]). In the analyses with the full program context (in Sections 4.4 and 5.3.3), these restrictions on fetches and context switches do not apply.

The source codes, binary codes, and CFGs presenting the worst-case paths, which are the baseline for the following WCET analyses of software synchronisation functions, are presented in the Appendices in Sections A.1, B.1, and C.1.

Example of CFG and Schematic CFG The Figure 4.4 shows an example of the CFG of a worst-case path exported from OTAWA, and a schematic representation of the CFG. Please note that the CFG presented in Figure 4.4 contains only basic blocks and no instructions, as the version with instructions is too big to be presented here in a readable size. The CFG with readable instructions is included in the Appendix C.1.

[5]For example, the TriCore architecture "can automatically save or store half the register context upon an interrupt within two cycles."(cf. TriCore 1.3 2002, p. 8)

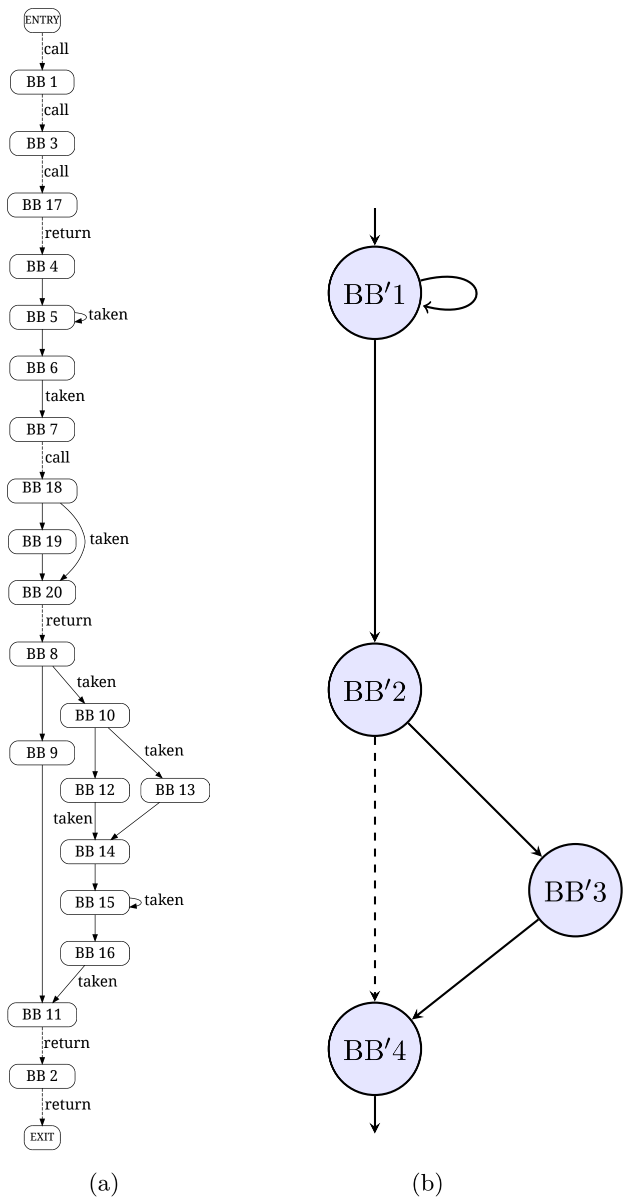


FIGURE 4.4.: Subfigure (a) shows as example an CFG of the worst-case path of the *lock()* function from the fair mutex lock implementation derived from OTAWA with basic blocks (BB 1 to BB 2) as overlay (original from .dot file including basic blocks and instructions can be found in the Appendix C.1), and the transformation of that CFG in a schematic CFG version in subfigure (b). The dashed edge in subfigure (b) (BB′2 to BB′4) shows a possible path in the execution that is not part of the worst-case path of the analysed thread (see details in Section 4.3.2).

Please note that a general CFG is even larger as it includes all possible paths; the ones displayed here only show the basic blocks and paths that are in the worst-case path. The Figure 4.4 displays the CFG of the worst-case path of the `lock()` function of the implemented fair mutex lock (see Section 3.3.3) derived from OTAWA in Figure 4.4(a). The Figure 4.4(b) shows the schematic CFG which was derived from the CFG from Figure 4.4(a). In the schematic CFGs some of the non-essential basic blocks and instructions are omitted or merged into one node, e.g. the ones that are specific to the TriCore-based MERASA ISA, are calls to functions, or inlined functions. For example, in the case depicted in Figure 4.4, the calls to functions have been merged (basic blocks BB 1 to BB 20 in Figure 4.4(a)) to one node BB′1 in the schematic CFG in Figure 4.4(b).

This allows for highlighting the more important relations, e.g. a while-loop representing a spin lock. The edges in the schematic CFGs show the control flow in the same way as in the original CFGs. Additionally, self-directed edges are added to represent while-loop structures which are used in nearly all `lock()` functions of the implemented busy-waiting software synchronisations (see node BB′1 in Figure 4.4(b)). However, please note that the behaviour of this while-loop from the spin lock is not visible in the original CFG (e.g. the one presented in Figure 4.4(a)), but in the flow facts. That is how often the jump for the while-loop is taken or not taken in the worst-case path depends on the number of competing threads and the length of the critical sections protected by this spin lock; it is part of the static WCET analysis after the CFG is constructed.

Another information that is included in the schematic CFGs are paths that are not part of the worst-case path of the analysed thread, but are in the worst-case path of competing threads. As the CFGs of synchronisations presented and analysed below are executed by all threads in concert, some of these threads execute a different path than the worst-case path of the analysed thread. This is highlighted by the dashed edges, e.g. in Figure 4.4(b) between BB′2 and BB′4 representing a possible path from the CFG in Figure 4.4(a) (the path BB8, BB9, and BB11). The dashed path is in the worst-case path of one of the other threads that are competing with the analysed one, and by that influence the WCET of the actual analysed thread. The reason why another thread executes the dashed path stems from semantic reasons, e.g. for the example of the mutex lock function in Figure 4.4(b): the worst-case path of the analysed thread includes that this thread does not gain the lock at first, and has to enter the waiting list. Hence, that implies that another thread (just) got the lock, and therefore executed the dashed path.

Another interesting detail from the blocking, fair mutex lock implementation is shown in the example in Figure 4.4(b) in the transition from node BB′3 to BB′4, or better, in the missing edge from BB′3 to BB′1. The edge BB′3 to BB′4 represents that in the fair mutex lock implementation the longest waiting suspended thread (BB′3) gains the lock after being unsuspended from the previous mutex owner (BB′4). A non-fair mutex lock implementation typically wakes all waiting threads, and they compete again for the lock (mostly also with other entering threads that have not yet been suspended and just reached the lock), hence this would be represented by an edge from BB′3 to BB′1. More details on the fair mutex implementation are presented in Section 3.3.3, and the static WCET analyses of the fair mutex lock implementations are presented in Section 4.3.2.

4.3.1. Busy-Waiting Synchronisations

In this section the three different software synchronisations using busy-wait (spinning) methods are analysed in detail.

A) Spin Locks with Test-and-Set

At a glance	
Pseudo code	Algorithm 3.1
Source code	Listing A.1
Binary code	Listing B.1
CFGs	part of Figure C.2
Schematic CFGs	Figure 4.5

The spin lock with TAS is a very simple synchronisation method to protect critical sections. It contains a TAS operation in a while-loop in its `lock()` function, and a TAS operation in its `unlock()` function (see schematic CFG in Figure 4.5). As already discussed in Section 3.3.1, the main problem of a spin lock with TAS is to assure fairness between competing threads, which in conclusion leads to problems determining the WCET. In the case of the MERASA architecture, it is mostly possible to analyse TAS-based spin locks due to the design of the memory interconnect. However, it is possible to construct situations in which fairness is, at least in the worst-case, not assured for every thread competing for a TAS spin lock and hence might experience unbounded delays. For the following analyses, it is assumed that such a behaviour does not occur, but, in general, it needs to be taken into account.

Lock Operation For the `lock()` function, the busy-waiting to gain the lock needs to be taken into account. Therefore, the WCET of the `lock()` function depends on the number of competing threads, and also on the WCWT which is defined by the length of the critical sections executed by the other threads.

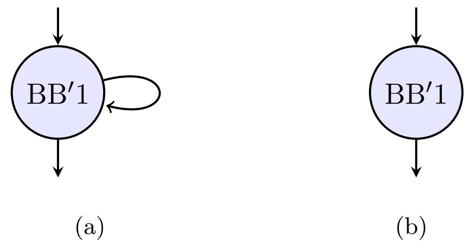


FIGURE 4.5.: Schematic CFG of the spin lock implementation with TAS: (a) shows the *lock()* function with a while-loop in which the TAS operation is executed, and (b) depicts the simple *unlock()* function which just contains a TAS (respectively a store) operation to release the lock.

The length of the critical section, and therefore also the corresponding WCET, might be variable for different threads; this depends on the program. Also, the WCET of the `unlock()` function executed by the other competing threads adds to the WCET of the `lock()` function.

Unlock Operation For the unlock operation, the WCET is easy to compute: As no busy-waiting is needed to unlock the spin lock, and only one memory operation is used in the `unlock()` function, the interference from other threads is included in the WCML of that TAS (or store) operation, and no waiting times need to be taken into account. Please note that it is possible to use a simple store operation instead of a RMW operation to unlock the TAS spin lock, but then it is more difficult for the OTAWA tool to locate the corresponding synchronisation point (cf. Section 3.3.1).

WCET Estimation In summary, to compute the WCET of one thread, the WCETs of `lock()` operation, critical section, and `unlock()` operation of all the other competing threads need to be added and the result forms the WCWT for the thread that is under analysis. Then, adding the WCET of this thread's `lock()` operation, critical section, and `unlock` operation sums up to the final WCET. In Section 4.3.4 the resulting WCETs of the `lock()` and `unlock()` function are shown.

B) Spin Locks with Fetch-and-Decrement

At a glance	
Pseudo code	Algorithm 3.2
Source code	Listing A.2
Binary code	Listing B.2
CFGs	part of Figure C.3
Schematic CFGs	Figure 4.6

The schematic CFGs of the `lock()` and `unlock()` functions of the spin locks with F&I/F&D, shown in Figure 4.6, are the same as for the spin lock with TAS. However, the implementation, that is the used atomic RMW primitive, is quite different. Please note that this spin lock implementation with F&D can only be used if the RMW operations F&D is implemented as in the augmented memory controller of the MERASA multi-core processor, that is a repetitive execution of F&D does not decrement any more when a given limit, e.g. '0', is reached (cf. Section 3.2.2).

Lock Operation For the WCET estimation of the `lock()` function executed by a thread, the competing other threads have to be taken into account in the same way as for the other busy-waiting lock with TAS: number of competing threads, and their WCETs of acquiring the lock, executing the critical section, and releasing the lock. So, the analysis is nearly the same as for the spin lock with TAS.

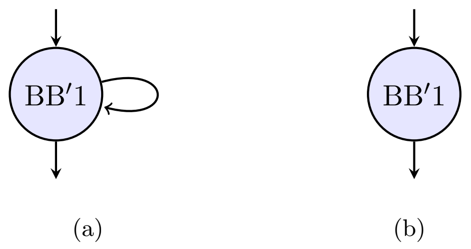


FIGURE 4.6.: Schematic CFG of the F&I/F&D lock implementation: (a) shows the *lock()* function containing a F&D operation in a while-loop, and (b) the *unlock()* function with just a F&I operation to release the lock.

Unlock Operation The `unlock()` function has been implemented with the F&I primitive. Nonetheless, the `unlock()` operation could be also implemented, as for the TAS spin lock, with a simple store operation. However, F&I has been chosen to analyse its impact on the WCET of an `unlock()` operation, and to allow the OTAWA tool to recognise it as a synchronisation operation. Also, the difference is not really big, as the F&I operation can be executed without waiting, so it shows the same behaviour as for an `unlock()` function with a simple store operation, and the WCET only differs in the WCML of the corresponding memory operation, and the number of instructions needed to prepare the memory operation.

WCET Estimation For the WCET estimation of spin locks with F&I/F&D, the same holds as for the spin locks with TAS. The results are shown and discussed in Section 4.3.4.

C) Ticket Locks with Fetch-and-Increment

At a glance	
Pseudo code	Algorithm 3.4
Source code	Listing A.3
Binary code	Listing B.3
CFGs	Figure C.1
Schematic CFGs	Figure 4.7

The CFG of the `lock()` operation for ticket locks is a bit different to the above discussed busy-waiting locks, as first a ticket is acquired, and then it is repetitively checked (spinning in BB′2) if the acquired ticket equals the actual served ticket. The `unlock()` CFG is the same as for F&I/F&D spin locks, as only the next served ticket id needs to be atomically incremented.

Lock Operation The `lock` operation of ticket locks is organized slightly different to the lock operations of other busy-waiting synchronisations.

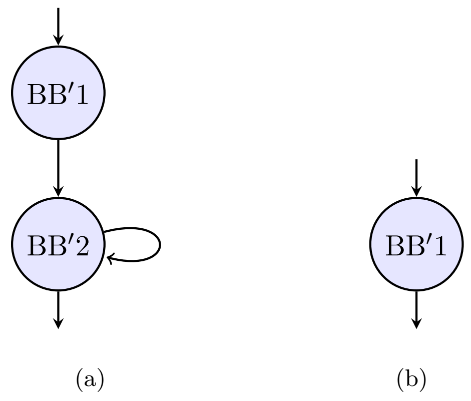


FIGURE 4.7.: Schematic CFG of the ticket lock implementation with F&I/F&D: (a) depicts the *lock()* function which acquires a new ticket and checks it against the actual served id; (b) presents the *unlock()* function which uses an F&I operation to increment the next served ticket id.

First, a ticket id is acquired with an atomic F&I operation. Then, the acquired local ticket id is repetitively checked against the global id of the ticket that is currently served. The first part of the locking operation—acquiring a ticket—is free of waiting introduced from competing threads (despite the impact captured in the WCML. Then, the spinning part (BB′2 in Figure 4.7) is analysed similar to the other busy-waiting locks.

Unlock Operation The WCET estimation of the `unlock()` operation of ticket locks is as simple as for the other busy-waiting locks: only one atomic F&I operation is used to increment the next served ticket id, therefore also no waiting time is introduced from executing the `unlock()` function. It would also be possible, but not recommended, to use a store operation to increment the next served ticket id, if the specific data structure needed for the F&I/F&D primitive is retained.

WCET Estimation In Section 4.3.4 the results of the WCET estimation of ticket locks are presented and compared to the other busy-waiting lock techniques.

WCET and Backoff Algorithms for Busy-Waiting Synchronisations

In the average-case, the use of a backoff algorithm is a common solution to reduce contention and/or increase throughput. For busy-waiting synchronisations, the idea of backoffs could help to reduce contention on shared resources. In an optimal case, the backoff impedes possible unsuccessful accesses or at least reduces them. Taking the hardware conditions into account, for example, it might be possible to only access the lock variables when they are “free”. Or, e.g. for ticket locks, a try-lock could check how much the actual served ticket id differs from the acquired own ticket id. If the difference is over a specific threshold, the thread could just wait/sleep for some time without spinning on the actual served ticket to reduce contention.

However, for worst-case situations this is a bit different. For example when analysing ticket locks with backoff algorithms, no impact on the WCET has been experienced. This, of course, highly depends on the chosen analysis technique. For the case of the static WCET analysis with OTAWA, interferences on the bus, that is contention, are already included in the analysis. Therefore, having one core producing less contention on the bus does not lead to better worst-case response times for the other cores (see also the discussion on busy-waiting versus blocking synchronisations in Section 4.3.4).

Beside the case of HRT, NHRT programs and best-effort execution can benefit from backoff algorithms by the reduction of ACETs. So, it might be promising to use backoff algorithms for NHRT programs, whereas for HRT programs the overhead from implementing backoff algorithms does not reflect in a reduction of the WCET.

4.3.2. Blocking Synchronisations

In this section the two implemented blocking synchronisations, (fair) mutex locks with TAS and binary semaphores with F&I/F&D, are discussed in detail.

A) Fair Mutex Locks with Test-and-Set

At a glance	
Pseudo code	Algorithm 3.3
Source code	Listing A.4
Binary code	Listing B.4
CFGs	Figure C.2
Schematic CFGs	Figure 4.8

The blocking synchronisations are different to the previous discussed busy-waiting synchronisations as they employ a waiting list and suspension of threads instead of busy-waiting with spinning. For the fair mutex lock implementation that means that acquiring and releasing the lock also needs to take into account acquiring the guard that secures the waiting list of suspended threads, that is a spin lock with TAS (BB′1 in both subfigures in Figure 4.8).

Lock Operation For the `lock()` operation, first a so-called guard, a spin lock with TAS in this case, needs to be acquired (BB′1 in Figure 4.8(a)). In the worst-case, the lock is already taken, and the thread under analysis is the last to reach the mutex lock, and therefore the last one that enters the waiting list (BB′3 in Figure 4.8(a)). The dashed path in Figure 4.8(a) (BB′2 to BB′4) shows the path that is executed by the thread successfully gaining the mutex lock. The WCET to enter the waiting list is similar to the WCET of a spin lock with TAS. But, the WCWT introduced from the length of the critical sections of competing threads is not only depending on the program code, but also on the code (e.g. in the RTOS) to manage the waiting list: it is known without program knowledge.

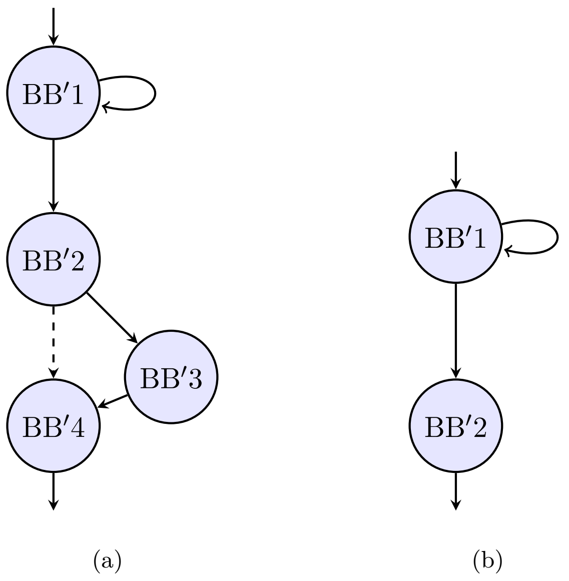


FIGURE 4.8.: Schematic CFG of the mutex lock implementation with TAS: (a) shows the *lock()* function, and (b) the *unlock()* function.

As waiting threads are suspended, their WCWT until acquiring the lock depends on the program code of critical sections, which is secured by the mutex lock, of the other competing threads. Also, the time it takes the thread to unsuspend after being woken must be included in the WCET of the `lock()` function.

Unlock Operation The worst-case situation for the `unlock()` operation of a (fair) mutex lock is more complicated than for the busy-waiting synchronisations, but still easy enough. The WCET includes the acquiring of the guard to the waiting list (BB′1 in Figure 4.8(b)), as the thread actually releasing the mutex lock first checks if already other threads are in the waiting list for the mutex lock. If so, the thread holding the mutex wakes the first thread in the waiting list, and releases the guard without freeing the mutex lock itself: it will directly be taken by the woken thread for strong fairness reasons discussed in Section 3.3.3. If no thread is waiting, it would set the mutex lock to be free, and would release the guarding spin lock. In the worst-case with N HRT threads, the thread under analysis competes with $N-2$ threads for the guard, and one thread is suspended and waiting, that is one thread entered the waiting list while the analysed thread held the mutex. The thread under analysis gets the guard after the other $N-2$ threads entered the waiting list, and then has to wake up the longest waiting thread (BB′2 in Figure 4.8(b)), and releases the guard.

WCET Estimation The results of the WCET analyses of the `lock()` and `unlock()` operations of the (fair) mutex lock implementation are shown in Section 4.3.4.

B) Semaphores with Fetch-and-Increment/Decrement

At a glance	
Pseudo code	Algorithm 3.5
Source code	Listing A.5
Binary code	Listing B.5
CFGs	Figure C.3
Schematic CFGs	Figure 4.9

In the presented analysis, the focus is on a binary semaphore implementation, not on the general case of semaphores. This allows for a comparison of the WCET estimates with the fair mutex lock implementation. The implementation of the binary semaphores is based on the FIFO waiting list with F&I (cf. Section 3.2.3), and on the special implementation of F&D as for the F&I/F&D spin locks (see also Section 3.2.2).

Lock Operation The lock operation, or `wait()` function as it is being called for the semaphore implementation, first checks if the semaphore is available. This resembles the behaviour of so-called try-locks, but is fully integrated in the `wait()` operation of (binary) semaphores. This relieves contention on the guarding F&D spin lock to reduce the WCWT.

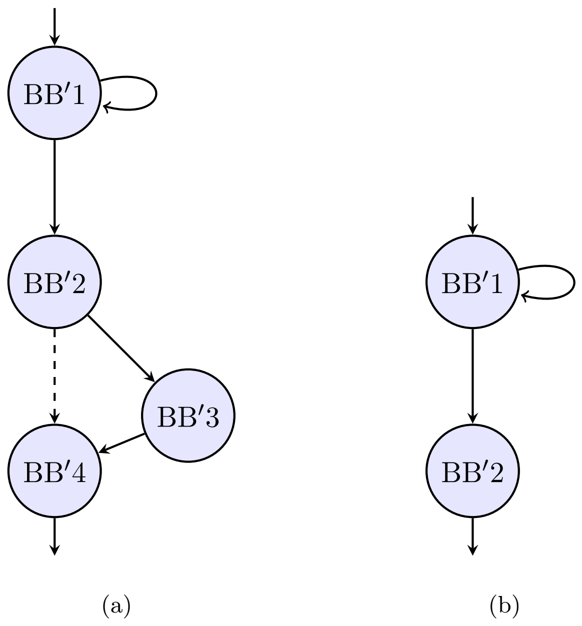


FIGURE 4.9.: Schematic CFG of the (binary) semaphore implementation with F&I/F&D: (a) shows the *wait()* operation, and (b) the *post()* function.

In the worst-case, the (binary) semaphore is not available so that the schematic CFG in Figure 4.9(a) starts with a guarding F&D spin lock in BB′1. This F&D spin lock guards the FIFO waiting list, and, in the worst-case, the analysed thread is the last to acquire the F&D lock and enters the waiting list (in BB′3 in Figure 4.9(a)).

As already described, the binary semaphore is very similar to the case of the fair mutex lock implementation. Thus, the dashed path from BB′2 to BB′4 in Figure 4.9(a) shows the same as for the (fair) mutex lock implementation, that is the path of the thread that successfully acquired the lock. The management of the FIFO waiting list with F&I/F&D is different from the management for the waiting list in fair mutex lock implementation, but the semantic is the same. So, for binary semaphores, the WCWT also depends on the length of the critical sections of the competing threads that are executed before the analysed thread (which in the worst-case means that all competing threads get the lock before the one under analysis). When the thread is finally woken and acquires the lock, it has to release the guarding lock (executing a F&I operation) which was previously acquired from the thread that has woken the thread under analysis.

Unlock Operation The `post()` function, that is the unlock operation of a semaphore, is semantically identical to the `unlock()` operation of the mutex lock. First, a thread executing `post()` needs to obtain the guard for the waiting list, and competes with all the other threads for it who are currently trying to enter it. Here, the same case as for the `unlock()` operation of the mutex lock holds: for N HRT threads, in the worst-case, one thread is actually waiting in the FIFO queue, while the other $N-2$ threads compete with the thread under analysis for the guard lock of the waiting list. When the analysed thread finally acquires the guarding lock (after the $N-2$ other threads, a second F&D operation is used to obtain the information if a thread is waiting in the FIFO queue. In the worst-case this is the case and the thread under analysis hands over the lock (without incrementing the semaphore counter) by waking the longest waiting thread. It does not release the guard lock, as this is done by the thread that was woken and now holds the lock. Else, if no thread is waiting, the actual thread uses two F&I operation to increment the semaphore counter, and releasing the guard lock. The worst-case stems from the execution when one thread is waiting, as then the thread under analysis has to compete with the $N-2$ other threads for the guard lock. Thus, in that case the analysed thread actually uses more F&I/F&D operations (5 F&I/F&D operations) as if the thread actually finds an empty waiting list (4 F&I/F&D operations), despite that the thread has to execute more code when a waiting thread has to be unsuspended (cf. pseudo code in Algorithm 3.5).

WCET Estimation In Section 4.3.4 results on the WCET analysis of binary semaphores are compared to the fair mutex lock implementations as two different blocking synchronisations, but as well the WCET results of the blocking synchronisations are compared with the WCET results of busy-waiting synchronisations.

4.3.3. Software Barriers

In this Section the WCET analyses of the two different barrier implementations, subbarriers with conditionals and F&I barriers, are described in detail.

A) Subbarriers with Conditionals

At a glance	
Pseudo code	—
Source code	Listing A.6
Binary code	Listing B.6
CFG	Figure C.3
Schematic CFG	Figure 4.10

The subbarrier implementation with conditionals has been introduced by Marejka (1994) to overcome the reinitialisation problem of typical barrier implementations with conditionals. Their major disadvantage for being used in HRT systems is the application of conditional variables (see discussion in Section 3.3.6), and the resulting high WCET.

Wait Operation The `wait()` operation of the subbarrier implementation uses a (fair) mutex lock when entering the barrier (merged in BB′1 in Figure 4.10). The same mutex lock is needed to secure the conditional wait (BB′3 in Figure 4.10) for threads waiting for the last thread to reach the barrier, thus satisfying the barrier condition. The last thread reaching the barrier needs to reset the current subbarrier and switch to the alternative subbarrier to overcome the reinitialisation problem.

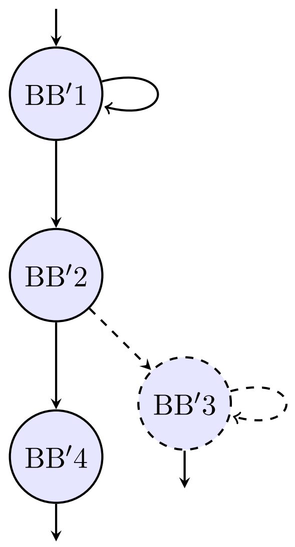


FIGURE 4.10.: Schematic CFG of the subbarrier implementation with conditionals.

Then it signals the other waiting threads to continue by using a conditional broadcast, and leaves the barrier (BB′4 in Figure 4.10). Thus, the last thread reaching the barrier defines the overall WCET of the `wait()` operation. The dashed edges (and BB′3 in Figure 4.10) show the execution path of the other threads that reach the barrier first, and enter the waiting list. FIFO ordering is not overly important for barrier implementations, as *all* waiting threads continue when the barrier condition (all threads reached the barrier) is fulfilled.

WCET Estimation The WCET estimation for a thread involved in a barrier with conditional variables is rather complicated. The reason is the conditional variable that is used for waiting threads needs the same mutex lock that secures the barriers (see details in Section 3.3.6). The subbarrier implementations overcomes this problem, as the last thread switches the subbarriers, and therefore also the needed mutex lock that guards the entry to the barrier. However, all this additional handling and logic causes a high WCET for the subbarrier implementation. The results of the WCET analysis of subbarriers are compared to the barriers that employ the F&I primitive in Section 4.3.4.

B) Barriers with F&I

At a glance	
Pseudo code	Algorithm 3.8
Source code	Listing A.7
Binary code	Listing B.7
CFG	Figure C.4
Schematic CFG	Figure 4.11

F&I barriers are a blocking implementation of barriers not prone to the reinitialisation problem, using the F&D spin lock, and a FIFO waiting list that is managed with F&I (as for the semaphores implementation as well, cf. Section 3.2.3).

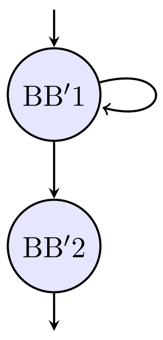


FIGURE 4.11.: Schematic CFG of the barrier implementation with F&I.

Wait Operation The `wait()` function of F&I barriers starts with a F&D spin lock, which secures the waiting list of the barrier. In the worst-case, the analysed thread reaches the barrier last, but just after all competing threads. Therefore, it has to wait to get the F&D spin lock, and then needs to wake up all waiting threads. Thanks to the specific implementation with cycling counting of F&I (see Section 3.2.3), the thread does not need to reset anything and continues after releasing the F&D spin lock. The global waiting queue is then back in the initial state, in which it was before any thread arrived at the barrier, hence being ready for the next iteration without being prone to the reinitialisation problem (cf. Section 3.3.6).

WCET Estimation For the WCET of barriers using F&I, the same holds as for the subbarrier implementation: the last thread reaching the barrier determines the overall WCET. As mentioned above, without taking the program structure into account, the last thread entering the barrier also competes with the other threads for the F&D spin lock guarding the barrier fulfilment check. Either the thread does not yet satisfy the barrier condition and enters the waiting list, or it does fulfil the barrier condition and wakes the other waiting threads. However, the program structure might lead to an even worse situation: depending on how much code the threads need to execute before reaching the barrier, the overall WCET might be higher due to waiting for the last thread to reach the barrier. In the analysis presented in Section 4.3.4, it is assumed that all threads reach the barrier at the same time and compete for the F&D spin lock, thus no waiting before reaching the barrier is assumed. For the WCET analysis of the parallel IFFT program in Section 4.4, the program code is included and therefore WCET estimates of different iterations secured by barriers might vary. However, the WCET guarantee computed without taking the program code into account is a lower bound for possible WCET estimates when the program code itself is included in the analysis, that is the different WCET estimates cannot be lower than the one computed in Section 4.3.4.

4.3.4. Results and Discussion

Wolf et al. (2010a) already presented WCET results on the preliminary implementations of software synchronisations in the MERASA RTOS, and Rochange et al. (2010) WCET results on a multithreaded program with these software synchronisations. For the results presented below, the integrated augmented memory controller, the new RMW operations (F&I/F&D), software synchronisations using these new RMW operations, and changes in the source code of software synchronisations fostering their timing predictability (e.g. for the mutex lock and barriers) have been taken into account (see also Gerdes et al. 2012b). Beside the changes, the methodology of the WCET analysis of the above software synchronisations is done as described by Rochange et al. (2010), Wolf et al. (2010a), and Gerdes et al. (2012b). Some of the formulas for the WCET estimations with OTAWA, using the computed WCMLs as presented in Section 4.2, have been provided by courtesy of Christine Rochange (University of Toulouse).

TABLE 4.2.:
WCET estimates of software synchronisations in the quad-core MERASA multi-core processor in number of cycles (load/store operations take five/four cycles, bus cycle time is one). Program dependencies are treated as a variable: $WCET_{\text{prg}}$.

Software Synchronisation	$WCET_{\text{exec}}$	$WCET_{\text{wait}}$	$WCET_{\text{total}}$
TAS lock()	63	$3 \cdot 63$	$\mathbf{252} + WCET_{\text{prg}}$
TAS unlock()	57	–	**57**
F&I/F&D lock()	64	$3 \cdot 64$	$\mathbf{256} + WCET_{\text{prg}}$
F&I/F&D unlock()	62	–	**62**
ticket lock()	267	$3 \cdot 220$	$\mathbf{927} + WCET_{\text{prg}}$
ticket unlock()	62	–	**62**
(fair) mutex lock()	1184	$3 \cdot 63$	$\mathbf{1373} + WCET_{\text{prg}}$
(fair) mutex unlock()	753	$2 \cdot 63$	**879**
binary semaphore wait()	658	$2 \cdot 64$	$\mathbf{786} + WCET_{\text{prg}}$
binary semaphore post()	357	$2 \cdot 65$	**487**
F&I barrier	592	$3 \cdot 64$	$\mathbf{784} + WCET_{\text{prg}}$
subbarrier	4640	$3 \cdot 63$	$\mathbf{4829} + WCET_{\text{prg}}$

The results presented in Table 4.2 show the estimated WCETs of the different software synchronisations split into the $WCET_{\text{exec}}$ of the execution itself, and the WCWT, that is the part of the $WCET_{\text{wait}}$ introduced from waiting. For the total WCET of some software synchronisations (lock and unlock operation), the program structure has to be included, which is shown by the summand in the last column of Table 4.2. This does not apply for the unlock operations, as the $WCET_{\text{prg}}$ is included in the corresponding lock operation. For those software synchronisations for that no WCWTs apply, e.g. the unlock functions of the busy-waiting synchronisations, this is denoted by a '–'. The evaluation settings for the WCET analyses, as far as not already mentioned in the caption of Table 4.2, are described in detail in Section 4.3. Please note that $WCET_{\text{prg}}$ is the WCET of the critical section secured by a lock, but for the barrier synchronisations $WCET_{\text{prg}}$ depends on the program code that is executed of each thread from the last synchronisation point until reaching the barrier. As already discussed in Section 4.3.3, it might be possible that the summand $WCET_{\text{exec}} + WCET_{\text{wait}}$ is actually lower, if $WCET_{\text{prg}}$ predominates it. However, it is not possible that a lower WCET guarantee as the depicted is achieved for the given implementations and evaluation settings.

The difference in the estimated WCETs of busy-waiting locks is rather small, at least for TAS and F&I/F&D spin locks. The only difference stems from the applied RMW operation: the WCML of a F&I/F&D operation is one cycle higher than for a TAS operation. Only for ticket locks, the `lock()` operation has a significant higher estimated WCET than the other two busy-waiting lock implementations.

The reason for this is that more logic needs to be done for a ticket lock compared to the other two, very simple busy-waiting locks: a ticket is fetched, the padding bits (the limit in the data word; see Section 3.2.2) need to be masked, and then the ticket lock operation spins on the actual served ticket id. The spinning does not need any RMW operations, but the padding bits must be masked each time the actual served ticket id is retrieved. Also, the ticket lock implementation is not perfectly tuned for efficiency on the TriCore-based MERASA architecture, therefore the waiting time of the ticket lock implementation includes five memory operations that increase the execution time; this is also due to the limited compiler optimisations used to foster the static timing analysability of the compiled code. However, the implementation is more robust for the programmer than the other busy-waiting operations, as those are usually only meant to be used in the system software and not explicitly by the programmer. The waiting time for all three busy-waiting implementations needs to be multiplied with the number of competing threads, that is three for the case of a quad-core processor with one HRT thread per core. NHRT threads, which might be also executed in concert with the HRT threads in the MERASA multi-core processor, do not compete for the same locks with HRT threads, as they are not allowed to share critical sections with HRT threads. However, it would be possible to allow such mixed-criticality execution, but it needs to be taken into account in the timing analysis. Even if the estimated WCET of the ticket lock implementation is higher than for the other busy-waiting implementations, it has the advantage of fairness mostly independent of the arbitration in the interconnect, and is therefore recommended to be used as busy-waiting spin lock (see Section 3.3.4).

The blocking binary semaphore implementation is very similar to the fair mutex lock implementation (cf. Section 3.3.5), but they differ by the RMW operations they internally rely on: the (fair) mutex lock implementation uses TAS spin locks while the binary (blocking) semaphore implementation is based on F&I/F&D operations. Another main difference is the way they secure the waiting list of threads to enter the critical section: the use of F&I/F&D operations makes it possible to implement hardware supported FIFO queues (see Section 3.2.3) that perform better, in the MERASA multi-core processor, than the software linked lists used in the (fair) mutex lock implementation (see also Section 3.3.3, and Gerdes et al. 2012b). Estimated WCETs in Table 4.2 show that the implemented semaphores are nearly two times faster than mutex locks for both the `lock()` and `unlock()` operation. The waiting times $WCET_{\text{wait}}$ to acquire a lock are depending on the number of concurrently running threads that compete for that lock.

In the evaluated case with $N = 4$ cores, this waiting time arises from one basic block being executed $N - 1 = 3$ times in the worst-case, as in the worst-case the other three threads gain the guarding spin lock before the thread under analysis. The only difference for the binary semaphores stems from the specific implementation of F&D in the augmented memory controller (cf. Section 3.2.2). By this, the semaphore `wait()` operation is enabled to first check if the semaphore counter is greater than 0, before competing for the waiting list with the other $N - 2 = 2$ threads that did not gain the lock (as one of the $N = 4$ threads must have successfully acquired the lock to enforce the worst-case). Also, for the `unlock()` operation of the (fair) mutex lock implementation respectively the `post()` operation of the binary semaphore implementation, this waiting time is only enforced by the $N-2=2$ other threads that can compete for the lock: one thread must be actually waiting in the waiting list to enforce the worst-case path for the thread releasing the lock (otherwise the analysed thread would execute a non-worst-case path without waking up a waiting thread).

In high-performance systems using blocking synchronisations is favoured over busy-waiting and spinning solutions to decrease traffic on the interconnect and hence contention on shared resources. Therefore, on the one hand, mutex locks and blocking semaphores are usually preferred over simple busy-waiting locks. On the other hand, busy-waiting locks that spin on local memories or use backoff strategies are a way to reduce contention and increase the average-case performance in high performance systems. But, those solutions are seen unsuitable for static timing analyses of parallelised HRT programs (see Sections 3.3.7 and 4.3.1), as contrarily to the ACET, the WCET of a thread is not impacted by the other threads' busy-waiting. So far it is impossible in a static WCET analysis of parallel programs on a multi-core processor to fully account for every operation of every thread on each core in every cycle. Thus, contention on shared resources, e.g. the memory interconnect, must be treated pessimistically and is included in the WCMLs. The WCMLs account for maximum possible interference on the shared memory and memory interconnect, that is all threads permanently issue requests to the main memory. So, backoff and suspension do not help to reduce a parallel program's WCET in the scope of HRT capable shared-memory multi-core processors.

As an alternative, the use of ticket locks as introduced in Section 3.3.4, which implement a busy-waiting strategy with F&I/F&D instructions, is recommended. Compared to mutex locks or blocking semaphores, the code of acquiring and releasing a ticket lock is shorter because it does not include inserting and suspending (respectively extracting) a thread in (from) a FIFO queue. Also, it seems to be feasible that critical section are mostly short in parallel HRT programs; this further reduces the benefit of suspension. In summary, worst-case efficient busy-waiting synchronisations are the medium of choice. The result is an improved WCET guarantee, as shown for two parallelised HRT programs in Section 4.4 and the WCET results in Table 4.2.

Table 4.2 also highlights a considerable improvement of WCET guarantees when using synchronisation barriers implemented with the F&I primitive instead of subbarriers based on conditions (*wait, broadcast*), as described in Section 3.3.6. The possibility of implementing FIFO queues in a very efficient way with F&I/F&D instructions already made the difference between the worst-case performance of binary semaphores versus the (fair) mutex lock implementation, and this is also the case for the F&I barrier implementation. The solution with F&I for barriers makes them, in the worst-case, more than six times faster than the subbarrier implementation. Both barrier implementations are not prone to the reinitialisation problem, i.e. the problem that threads reentering the barrier while other threads have not left it yet after the previous round. Hence, F&I barriers are sufficiently robust and worst-case efficient alternatives to subbarriers. Please note that while the WCET for F&I barriers is rather low compared to mutex locks, a major portion of the WCET could be included in the $WCET_{\text{prg}}$ part, depending on the program code. Further results on the improvement of WCET guarantees with F&I barriers are presented in the following Section 4.4.

4.4. WCET Analyses of Parallelised HRT Programs

In this section the static WCET analysis with OTAWA of two parallelelised HRT programs is shown: *matmul* and *IFFT*. The *matmul* program is a simple parallel matrix multiplication which secures access to next computations by locks, whereas *IFFT* is a parallel, integer version of a Fast-Fourier-Transformation (FFT) using locks to secure shared data and barriers to separate iterations.

4.4.1. matmul: Parallelised Matrix Multiplication

The *matmul* program is a parallelised matrix multiplication being used as a benchmark for the different locking software synchronisations. Thus, it was chosen to use a naïve parallelisation in which the dispatching of work units—each computing one result cell in the result matrix (see Figure 4.12)—to threads is secured by a critical section.

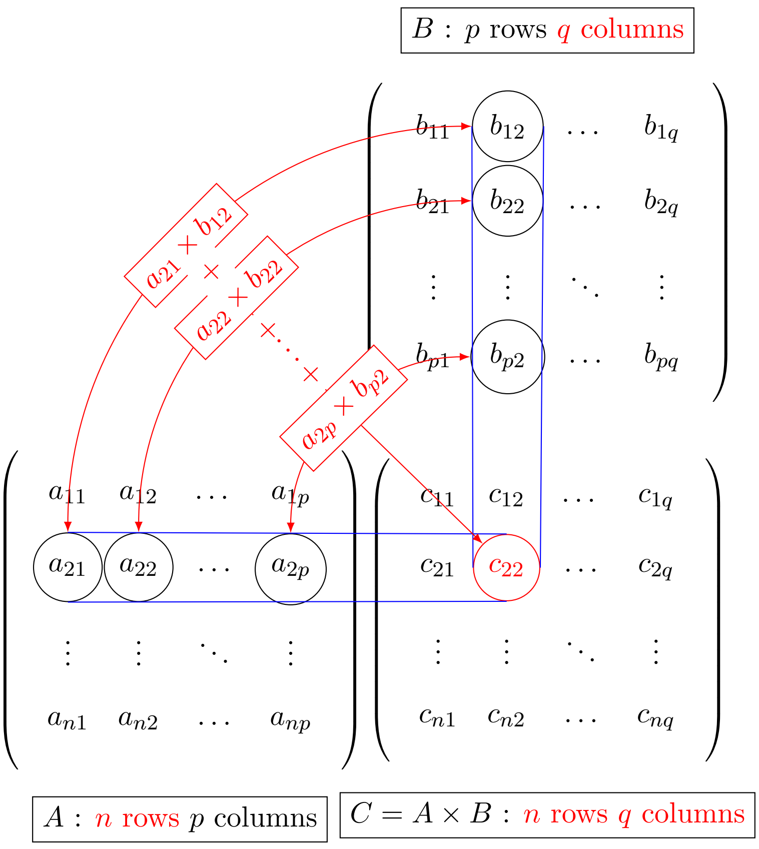

B : p rows q columns
b_{11} b_{12} … b_{1q}
b_{21} b_{22} … b_{2q}
b_{p1} b_{p2} … b_{pq}
$a_{21} \times b_{12}$
$+$
$a_{22} \times b_{22}$
$+ \dots +$
$a_{2p} \times b_{p2}$
a_{11} a_{12} … a_{1p}
a_{21} a_{22} … a_{2p}
a_{n1} a_{n2} … a_{np}
c_{11} c_{12} … c_{1q}
c_{21} c_{22} … c_{2q}
c_{n1} c_{n2} … c_{nq}
A : n rows p columns
$C = A \times B$: n rows q columns

FIGURE 4.12.: Falk's scheme of the naïve parallelisation of matmul. Every cell of matrix C is secured by a critical section and computed in one step by one thread. *(TikZ source for this figure by courtesy of Alain Matthes, see `http://altermundus.com/`)*

Table 4.3.:
WCET estimates (# cycles) of synchronisation techniques in *matmul* executed on a quad-core MERASA processor derived from OTAWA with $T_{\mathrm{L}} = 5$ cycles respectively $T_{\mathrm{L}} = 10$ cycles memory latency for a normal load, and one cycle bus latency.

Configuration	Mutex Lock	(Binary) Semaphore	Ticket Lock
4 cores, $T_{\mathrm{L}} = 5$	1,589,483	1,221,477	1,044,762
4 cores, $T_{\mathrm{L}} = 10$	2,635,908	2,119,687	1,833,392

These locking mechanisms were substituted for each separate WCET analysis, namely the (fair) mutex lock, the ticket lock, and the binary semaphore implementations have been used for the comparison.

In the computation of $A \cdot B = C$, the input matrices A and B can be seen as read-only, therefore no synchronisation is needed when reading from them. As the work units are dynamically dispatched to a thread, and each working unit is only distributed once to a thread, writing the results to the result matrix C is automatically free of collisions. The *matmul* program has been chosen as a benchmark for the implemented synchronisations, thus the focus is not on the efficiency of the parallelisation, but on the worst-case performance of the different implemented software synchronisations.

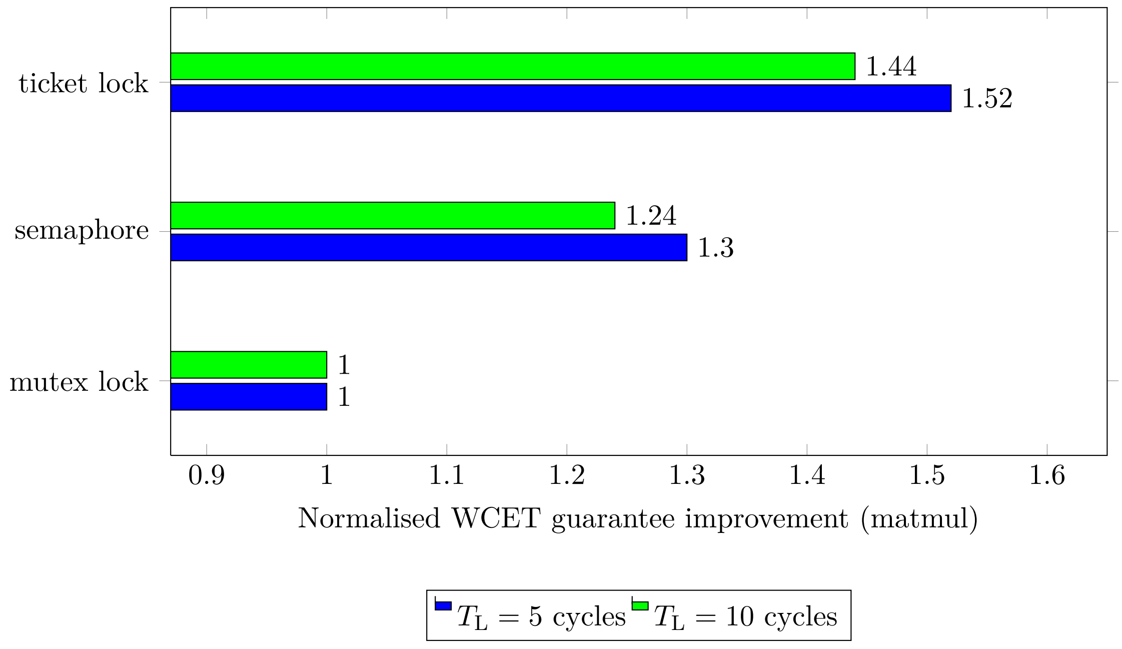


Figure 4.13.: WCET guarantee improvements on a quad-core MERASA processor with five and ten cycles latency for a load operation for the parallelised matrix multiplication (*matmul*) using three different software synchronisations normalised to the configuration with mutex locks.

Table 4.3 shows the WCET estimates for matmul computed by OTAWA with two configurations, that is four cores and five respectively ten cycles memory latency for a load operation. The WCMLs have been derived from the equations in Section 4.2.

The improvements of WCET guarantees depicted in Figure 4.13 are normalised on the version with (fair) mutex lock for both configurations (with either five respectively ten cycles load latency). The results in Table 4.3 and Figure 4.13 show that ticket locks perform better than the binary semaphore and the mutex lock implementations for all configurations. In detail, the improvement of WCET guarantees is 1.44 respectively 1.52 for ticket locks compared to the version with (fair) mutex locks. The comparison in Figure 4.13 also shows that the difference in the gain for the WCET improvements is nearly the same for five cycles and ten cycles load latencies: the difference in the gain is smaller than 6 %. These results support the findings from Section 4.3.4, which show that busy-waiting synchronisations perform better than blocking ones, that is ticket locks outperform the two blocking software synchronisation techniques.

4.4.2. IFFT: Parallelised Integer Fast-Fourier-Transformation

The *IFFT* program has been parallelised for the MERASA processor in a master thesis by Eser (2010). As baseline for the parallelisation, an iterative version of the radix-2 algorithm was used. The algorithm works *in place* and the sample points are stored in an array. The iterative version maps the floating-points values from $[-1, 1]$ on fixed-point numbers in the range $[-32767, 32767]$. Depending on the number of sample points N, the *IFFT* program computes results in one initialisation and $\log_2(N)$ calculation steps.

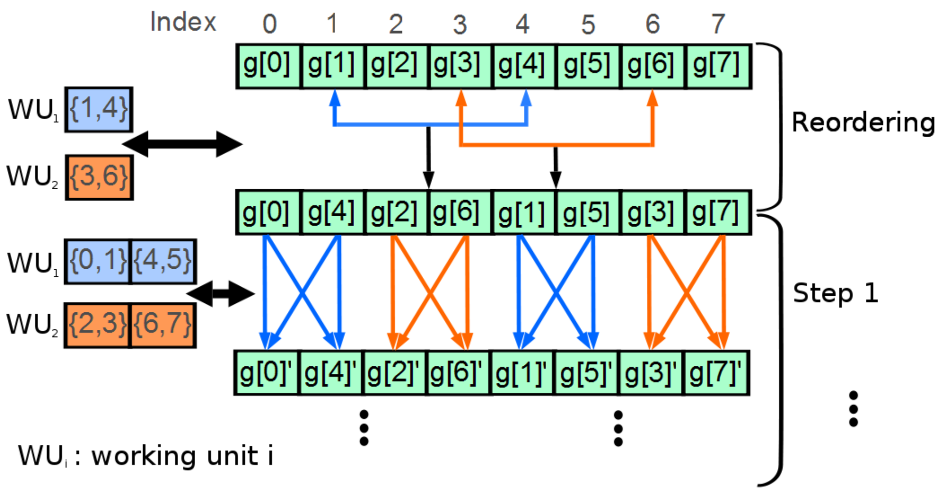

Index
0
1
2
3
4
5
6
7
g[0] g[1] g[2] g[3] g[4] g[5] g[6] g[7]
WU1 {1,4}
WU2 {3,6}
Reordering
g[0] g[4] g[2] g[6] g[1] g[5] g[3] g[7]
WU1 {0,1} {4,5}
WU2 {2,3} {6,7}
Step 1
g[0]' g[4]' g[2]' g[6]' g[1]' g[5]' g[3]' g[7]'
WUi : working unit i

FIGURE 4.14.: Parallelisation of IFFT: steps are separated by different barrier implementations, and dispatching working units to threads is secured with different locking techniques for each single WCET analysis. (Image is a slightly changed version taken from Eser (2010))

Table 4.4.:
WCET estimates (# cycles) of different synchronisation techniques in *IFFT* executed on a quad-core MERASA processor derived from OTAWA with $T_L = 5$ cycles resp. $T_L = 10$ cycles memory latency for a normal load, and one cycle bus latency.

(a) IFFT Conditional Subbarriers

Configuration	Mutex Lock	(Binary) Semaphore	Ticket Lock
4 cores, $T_L = 5$	288,781	224,196	204,933
4 cores, $T_L = 10$	463,756	370,626	339,798

(b) IFFT F&I Barriers

Configuration	Mutex Lock	(Binary) Semaphore	Ticket Lock
4 cores, $T_L = 5$	209,572	135,420	120,817
4 cores, $T_L = 10$	335,217	224,345	202,752

In the initialisation step, the sample points are reordered for the following calculation steps. In each calculation step, partial solutions are computed and combined by one to $\frac{N}{2}$ working units. The working units can be computed in parallel, but the distribution of a fresh working unit to a thread is secured by a lock. Figure 4.14 presents the process of reordering and combining in the parallelised *IFFT* program as done by Eser (2010) with two working units. All steps are separated with barriers, that is either the subbarrier or the F&I barrier implementation (see details in Section 3.3.6). Combining of results in the calculation steps is scheduled dynamically to the threads, and secured by critical sections with either (fair) mutex locks, binary semaphores, or ticket locks.

The WCET estimates from OTAWA are depicted in Table 4.4 for the different configurations with the two different barrier implementations respectively the three locking mechanisms, and for the same configurations as for the *matmul* program above, that is a four-threaded version with five respectively ten cycles load latency. As reference for the comparison of the chosen barrier and locking implementations in Figure 4.15 the slowest configuration of the parallelised *IFFT* program with subbarriers and mutex locks has been chosen for both configurations, that is a baseline of 288,781 cycles for the version with five cycles load latency, and 463,756 cycles for the version with ten cycles load latency. The normalised improvements of WCET guarantees show that the version with ticket locks and F&I barriers performs best in the worst-case, reaching an improvement of WCET guarantees of 2.39 for five cycles load latency, respectively 2.29 for ten cycles load latency. The improvements of the WCET guarantees with ten cycles load latency are again close to the ones with five cycles load latency: the difference is less than 5 %.

Summing up, the results show again that ticket locks are very efficient in the worst-case, and additionally that F&I barriers perform much better than subbarriers with conditionals, as already assumed from the WCET results of software synchronisations in Section 4.3.4.

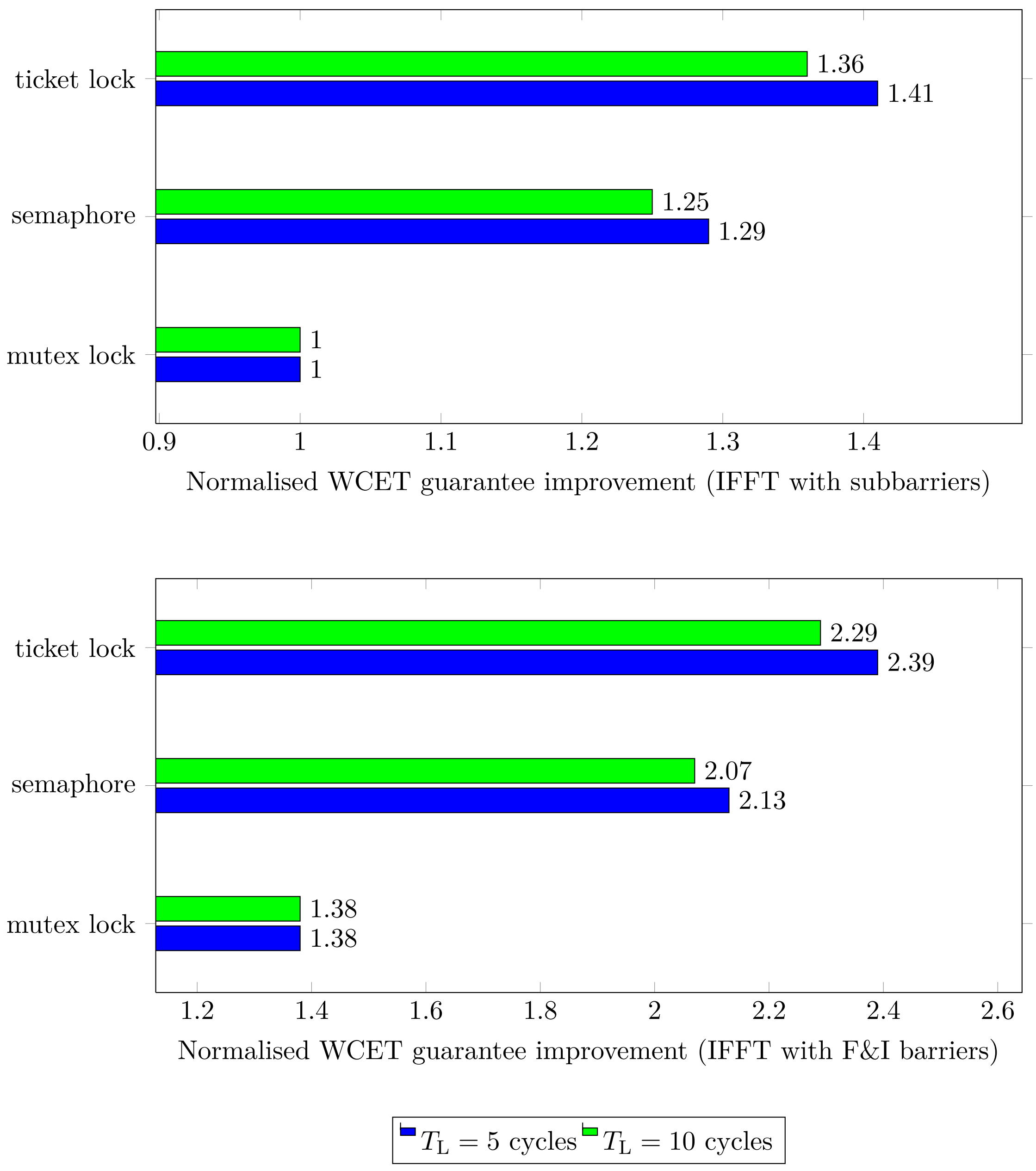


FIGURE 4.15.: WCET guarantee improvements on a quad-core MERASA processor for the parallelised *IFFT* with **subbarriers** (top) and **F&I barriers** (bottom) using different software locking techniques, and a load memory latency of five respectively ten cycles. The improvement of WCET guarantees are normalised to the configuration with (fair) mutex locks and subbarriers.

4.5. Related Work

Timing predictability is defined diverse in literature: Thiele and Wilhelm (2004) define timing predictability as the pessimism in the WCET, respectively in the BCET. That definition is also used in this thesis. Wilhelm et al. (2008) defines *(timing) predictability* as the difference between the *lower bound* and *upper bound* of a system, or, in other words, how much variation between the estimated best- and worst-case in a system is possible. In this thesis, this definition will be used as *system predictability*, however, Wilhelm et al. (2008) state that not only the system and architecture influences the predictability, but also the method or WCET tool that were used to compute or estimate such lower and upper bounds. The terms *timing analysability* and *timing predictability* are defined as *safety* for timing analysability, and *precision* for timing predictability by Wilhelm et al. (2008). A further definition of *timing predictability* is given by Kirner and Puschner (2010); they summarise different definitions and define timing predictability as based on the model used in the WCET analysis. A short survey on timing predictability has been recently published by Schoeberl (2012).

To estimate WCMLs, a predictable arbitration scheme for shared resources, for example a shared memory, is mandatory. The WCET model of the MERASA multi-core processor used in this thesis maintains predictable arbitration by a round-robin arbitration in the bus interconnect (Paolieri et al. 2009a, Ungerer et al. 2010). In a recent publication, Yoon et al. (2011) refined round-robin arbitration proposing a *harmonic round-robin* arbitration. In that way, memory intensive programs are given access to the bus more frequently by prioritising them in the bus scheduling. Further approaches for predictable bus arbitration using a TDMA scheme are presented by Andrei et al. (2008), Schranzhofer et al. (2010). Staschulat et al. (2007) state a different method for estimating upper bounds for memory latencies by linking task- and system-level analyses.

A number of publications (see e.g. Cullmann et al. 2010, Ferdinand et al. 2001, Hansson et al. 2009, Hardy et al. 2009, Heckmann et al. 2003, Kelter et al. 2011, Mohan et al. 2011, Puschner and Schoeberl 2008, Rosen et al. 2007, Wilhelm et al. 2009a,b, Yan and Zhang 2008) target timing analyses of multi-core processors, especially shared cache memories, scratchpad memories, (shared bus) interconnects, and memory controllers, but without the scope for multithreaded parallel programs.

Chattopadhyay et al. (2012) introduce a unified WCET analysis framework for multi-core platforms, extending their previous presented work (see Chattopadhyay et al. 2010). They start with a multi-core processor build from a simple-scalar processor (see Austin et al. 2002) with shared cache memories and a shared bus. For the WCET analysis they employ the chronos tool (cf. Li et al. 2007) in an updated version for multi-core processors. They analyse multiprogrammed (singlethreaded) workload, that is the Mälardalen WCET benchmarks[6]. Chattopadhyay et al. (2012) focus on the microarchitectural influences of the pipeline and cache on the estimated WCET, but do not cover multithreaded parallel program execution so far.

[6]see http://www.mrtc.mdh.se/projects/wcet/benchmarks.html for the source code and details on the Mälardalen WCET benchmarks [last accessed: April 2013]

Li et al. (2009) present WCRT analyses of a concurrent program running on shared-memory multi-core processors with a shared instruction cache. The communication is handled by message passing, and shared-memory synchronisation is not examined. Li et al. (2009) focus on the interference on a shared cache of multiple tasks of one application, and they derive WCRTs for different cache policies on abstract representations—*Message Sequence Charts* (MSC)— that allow for introducing communication between tasks.

The problem of (static) timing analyses of multithreaded parallel HRT programs has only recently started to get more attention in the research community, for example Lisper (2012) presents a short survey on the analysability of different parallel programming models for HRT systems, e.g. bulk synchronous parallel (BSP) (cf. Skillicorn et al. 1997) and CUDA[7].

An example that incorporates predictable architectures, that is the precision timed (PRET) architecture (cf. Edwards and Lee 2007), for predictable programming for HRT embedded systems is presented by Lickly et al. (2008). They also present how e.g. mutual exclusion can be handled in their approach. On the one hand, they argue that accesses to memory are always atomic by the use of a *memory wheel* implementation in PRET architectures. On the other hand, they apply *timing instructions* to enforce correct ordering of memory accesses for mutual exclusion. That is by providing distinct offsets as deadline at the beginning of the program, Lickly et al. (2008) enforce that e.g. a read and a write operation of a consumer-producer communication are executed in different rotations of the PRET architecture's memory wheel.

Gustavsson et al. (2010) present the chain of a possible static WCET analysis of multi-core architectures. They use timed automata to model the various components of a multi-core architecture, including private and shared caches, but also software-level shared resources like spin locks. The WCET of a program is then derived by model checking. In a later publication, Gustavsson et al. (2012) then present a possible static timing analysis of parallel HRT programs using abstract interpretations. So far, they introduce and proof their concept on those abstract representations, but yet no locks in parallel programs have been studied.

Mittermayr and Blieberger (2012) present a framework to analyse concurrent programs using so-called concurrent program graphs (CPGs). The CPGs are created from *Refined Control Flow Graphs* which, in turn, are based on CFGs. Then, the timing analysis is done using a dataflow analysis approach based on previous work by Blieberger (2002). In contrast to most other static WCET analysis frameworks, which start from representations of the binary code, their framework starts from source code representation. However, taking into account that the underlying timing analysis abstracts from microarchitectural properties (pipeline, caches, memory hierarchy etc), this might be reasonable for the approach, but most likely not practical. The approach seems overly pessimistic in that case and might lead to highly overestimated upper bounds. This influence of a compiler, thus the differences in source and binary code representations, and the microarchitectural properties are not discussed by the authors.

[7]More details on CUDA are available at NVIDIA's website at `https://developer.nvidia.com/category/zone/cuda-zone` [last accessed: April 2013]

But, the abstraction from microarchitectural properties and the use of a source code representation might lead to non-safe upper bounds. That is in the case of timing anomalies (cf. Reineke and Sen 2009), but also depending on properties of the pipeline, local instruction and data memories, the memory interconnect, shared resources, etc. Then, the requirement of a constant timing of a basic block execution, as assumed in the approach presented by Blieberger (2002), might not hold. Thus, it is possible that the control flow and data flow changes with variable execution times of one basic block, and that the abstract representation constructed from source code representation is not sound. Also, the authors present a proof for an example with two threads and a semaphore secured communication between them in a theoretical way. It is questionable if such a representation could actually cover the interaction of atomic hardware primitives and software synchronisation techniques in practice.

Wolf et al. (2010a, 2011) introduce the basic principles of analysing the WCWTs in synchronisation functions. The idea is to determine all the paths on which a thread holds any system-level or application-level synchronisation variable, and their estimated WCETs are combined to compute the WCWTs at synchronisation points. Rochange et al. (2010) present first results on the static WCET analysis of an industrial, parallel HRT program with a static WCET analysis. Rochange et al. (2010) consider a limited set of synchronisation functions based on the *swap* instruction. The grain of the parallelism in their program is more coarse-grained than in the programs considered in this thesis so that the cost of synchronisations compared to the computation time is relatively small. Gerdes et al. (2012b) then extended and further investigated predictable HRT capable implementation of common software and hardware synchronisation techniques, and their impact on the program's WCET. Gerdes et al. (2012b) use TAS, and F&I/F&D as hardware primitives, and mutex locks, semaphores, and barriers as software synchronisation techniques. Chapter 3 (and parts of this chapter) are based on that publication, but show further extensions and refinements. Gerdes et al. (2011) present WCET results on a parallelised large drilling machine control code using the commercial, measurement-based WCET tool *Rapitime* (see Rapita Systems Ltd. 2011).

Brandenburg et al. (2008) investigated in a survey real-time resource sharing in multiprocessors with lock-based and non-blocking techniques. They compare spinning busy-waiting locks against blocking (suspension-based) locks against lock-free and wait-free algorithms. The study has been conducted on their Linux test-bed called LITMUS^{RT} (LInux Testbed for MUltiprocessor Scheduling in Real-Time systems) on a COTS symmetric multiprocessing (SMP) system (four cache-coherent Intel Xeon^{TM} processors). They conclude that, concerning resource sharing, suspension-based locking is "*never* preferable" over busy-waiting locking using state-of-the-art analysis techniques, and if not more than 20 % of the entire program is spent in critical sections (which Brandenburg et al. (2008) state to be highly unlikely). Although the study is based on measurements on the one hand, and on a system that is not sufficiently predictable to be used in real-time systems on the other hand, the results are similar to the findings in this thesis for synchronisation in multithreaded parallel HRT programs on a predictable shared-memory multi-core processor.

Time is what keeps everything from happening at once. – Raymond King Cummings

5 Split-phase Synchronisation Technique

The idea of the split-phase synchronisation technique is to speedup the very frequent loads from a slow (off-chip) memory by a technique that slightly defers the infrequent synchronisation (RMW) accesses. As detailed in the chapters before, a major ratio of pessimism is introduced from using WCMLs in the WCET analysis, that is including overly pessimistic interferences on shared resources in the static analysis. The split-phase synchronisation technique alleviates the effect of slower memory operations in the execution itself, however, the main advantage stems from its effect in the static WCET analysis of parallel programs, that is a vast reduction of pessimism for the WCMLs of faster memory operation, thus, a reduction of the overall estimated WCET.

As stated by Cullmann et al. (2010):
“[It] is an open problem how to limit the information loss about concurrently running tasks by suitable abstractions. Hence, limiting inherent interferences must be a high-priority design goal: if there can be no interferences at all in the concrete system, it is easy for an analysis to exclude interferences even when abstracting completely from other tasks. One first principle for predictable architecture design is to strive for a good compromise between cost, performance, and predictability, concerning the sharing or duplication of resources.”.

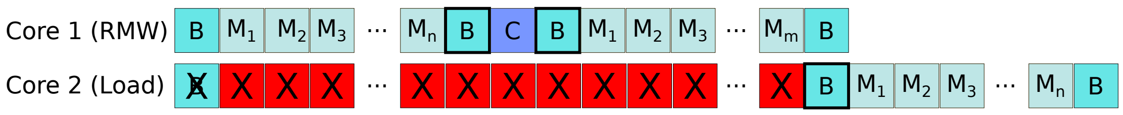
In the case of analysing the timing of parallel programs executing on multi-core processors, one main problem is to estimate the WCET of a concurrent code region in which synchronisation functions are used. In a static WCET analysis it has to be assumed that faster memory accesses like data loads, data stores or instruction fetches, which are usually the most frequent memory operations in a program, are executed with concurrent slower synchronisation operations of other cores. Thus, this core suffers from pessimism for every memory operation in the WCET analysis. One well-known solution to reduce the latency of memory operations from a global memory is duplicating data to local memories. In the case of the MERASA multi-core processor, this is done through timing predictable scratchpad memories (see e.g. Metzlaff et al. 2011, Ungerer et al. 2010, Whitham and Audsley 2009). It is, however, also a well-known fact that not all data can be replicated to local memories without reaching memory space limitations or increasing the latencies of local memories. Therefore, in this chapter, the split-phase synchronisation technique is introduced as an alternative solution to duplicating shared resources. The split-phase technique achieves a reduction of the pessimism in the WCET by splitting atomic RMW operations into three phases (load phase, modification phase, and store phase). This allows for execution of other memory operation, e.g. load operations, in between these phases. It is important that data consistency and the atomicity of the split RMW operation is not invalidated by an advanced memory operation that is executed in between these phases. In Section 5.2.4 it is shown that atomicity and memory consistency are adhered with the *weak consistency* model of Adve and Hill (1990).

Please note that this chapter presents extended work on the split-phase technique based on a publication at the *18th IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)* (see Gerdes et al. 2012a).

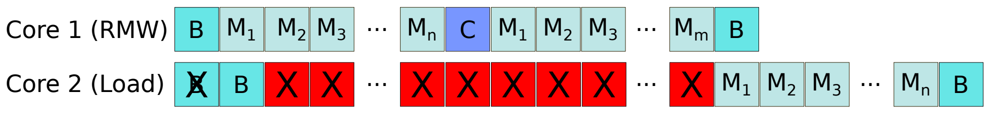
In Section 5.1, different solutions for atomicity of RMW operations as mandatory primitives for software synchronisations are discussed, including locking of the complete interconnect and implementing synchronisation logic outside of the core, e.g. as the augmented memory controller (introduced already in Section 3.2). In Section 5.2 a possible implementation of the split-phase synchronisation technique is presented. The aim is to achieve a timing predictable, and atomicity/consistency maintaining implementation of the split-phase synchronisation technique in the augmented memory controller. The split-phase synchronisation technique is independent of the MERASA architecture: it is adaptable for being used in other shared memory multi-core processors as well.
Section 5.3 presents an evaluation of the split-phase synchronisation technique. Firstly, the computation of WCMLs with the split-phase technique is shown (see also Section 4.2), and the impact on WCET guarantees is quantified. Concluding, results of the WCET computation for two parallelised HRT programs are presented to show the beneficial impact of the split-phase synchronisation technique on WCET estimates computed with a static timing analysis tool.

5.1. Introduction to the Split-phase Synchronisation Technique

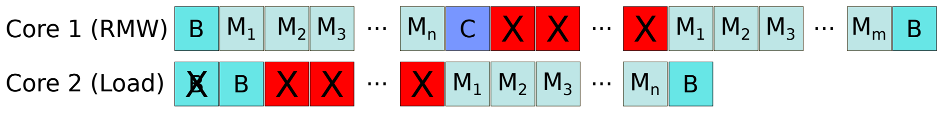
In the augmented memory controller, all upcoming requests are queued and executed in FIFO ordering maintaining fairness between cores by the arbitration strategy of the memory bus. Accesses are categorised into single memory accesses like a load or a store, and in RMW accesses on synchronisation variables composed of a load, a possible data modification, and a subsequent store. The RMW accesses are split in a load/modification phase and a store phase, and the memory address of a RMW access is buffered. If consecutive RMW accesses request the same memory address, that is the same synchronisation variable, the actual RMW access needs to complete the store phase before the next RMW access begins its load phase to retain correctness. Also, single load and store operation might access synchronisation variables (see e.g. Algorithms 3.1, 3.2). Therefore, they are also not allowed to interrupt an ongoing RMW operation, that is when the load phase of the RMW operation has already started. In case of an upcoming memory access to a different address, that is a different variable then the synchronisation variables accessed before, that access can be executed in between the load and store phase of the previous access without violating data consistency and atomicity of the synchronisation primitive. Then, to achieve reduced WCMLs for normal loads/stores, memory operations are reordered in the augmented memory controller. Load/store operations are prioritised over RMW operations, while keeping consistency with weak ordering as defined by Adve and Hill (1990). Additionally, an extra data bit, the *reorder flag*, is attached to every memory operation. The *reorder flag* prevents RMW operations from being delayed infinitely, while also guaranteeing the best possible worst-case for the prioritised load/store operations. In summary, the split-phase synchronisation technique allows to reduce the WCMLs for faster memory accesses, e.g. an instruction fetch or data load, but with the sacrifice of an increased impact on the WCMLs of RMW operations.


Core 1 (RMW)
Core 2 (Load)
B M1 M2 M3 ··· Mn B C B M1 M2 M3 ··· Mm B
X X X X ··· X X X X X X X ··· X B M1 M2 M3 ··· Mn B

(a) Locked interconnect


Core 1 (RMW)
Core 2 (Load)
B M1 M2 M3 ··· Mn C M1 M2 M3 ··· Mm B
X B X X ··· X X X X X ··· X M1 M2 M3 ··· Mn B

(b) Augmented memory controller


Core 1 (RMW)
Core 2 (Load)
B M1 M2 M3 ··· Mn C X X ··· X M1 M2 M3 ··· Mm B
X B X X ··· X M1 M2 M3 ··· Mn B

(c) Augmented memory controller with Split-Phase Synchronisation

FIGURE 5.1.: Memory access pattern for RMW operations with a locked interconnect (a), with the augmented memory controller (b), and with the split-phase synchronisation technique implemented in the augmented memory controller (c).

5.1.1. Discussion on Solutions for Atomicity of RMW Operations

The Figures 5.1(a) and 5.1(b) have been already discussed in Section 3.1.4. Including the split-phase synchronisation technique and its impact on the latency introduced to competing cores on the memory interconnect, presented in Figure 5.1(c), a comparison of the three different approaches is done.

As already discussed in Section 3.1.4, the augmented memory controller reduces the WCML for every memory access in comparison to a solution with a locked interconnect. However, due to needed pessimism in the WCET analysis to compute safe upper bounds, the WCML of normal load and store operations are still high, as it is assumed they compete with concurrent (slower) RMW operations at the bus interconnect to access the shared memory. This pessimism in the static timing analysis can be reduced by applying the split-phase synchronisation technique.

Figure 5.1(c) shows the reduction of the WCMLs of normal loads and stores, as they are allowed to execute in between the load and store phase of a RMW operation. In detail, the store phase of the RMW operation in *Core 1* is deferred (red boxes with 'X' in Figure 5.1(b)) by the split-phase synchronisation technique, and executed in the memory controller after the load in *Core 2* is finished. Thus, normal load and store operations, that are not accessing the same synchronisation variable as memory operations dispatched before them, suffer less latency from competing RMW operations. Further discussions on the impact of the split-phase synchronisation technique on the WCMLs and the estimated WCETs of parallel programs are presented in Section 5.3.

5.2. Implementation in the Augmented Memory Controller

The split-phase synchronisation technique has been implemented in the augmented memory controller of the MERASA multi-core processor. In detail, the RMW operations are split into three phases: a load phase, a modification phase, and a store phase. Other memory operations that do not access the same variable are allowed to be brought forward and executed before the store phase of the RMW operation. The target of the split-phase synchronisation is to achieve WCMLs for loads/stores that are, in a manner of speaking, the best possible worst-case. That means that the WCMLs of loads/stores only depends on concurrent (fast) loads/stores and not on concurrent (slower) RMW operations from other cores. Memory requests are handled as described in Section 3.1.1. For the split-phase synchronisation, further hardware changes in the augmented memory controller are needed to allow reordering while preserving atomicity (see Section 5.2.4).

The following proposed implementation does not claim to be the best possible technical solution. Further enhancements might decrease the needed logic and space, or even increase the average-case performance. However, from the worst-case timing analysis perspective the aim is to proof that a working technical implementation is possible that fulfils the requirements of consistency and atomicity for the split-phase synchronisation technique. Therefore, the focus is not on the details and size of the technical implementation, but on the approval of predictable worst-case timing.

The proposed hardware implementation in the augmented memory controller uses two register files as FIFO buffers for memory requests (see Figure 5.2). One register file, the *mem_buffer*, is used to store all memory requests, whereas the other register file, the *reorder_buffer*, is used as a temporary buffer to reorder the load/store requests of split RMW operations and load/store accesses on synchronisation variables. Also, an additional buffer (*sync_buffer*) is used to store synchronisation variables and a counter for each ongoing synchronisation access. Synchronisation accesses are either RMW operations, or also load/store operations on synchronisation variables, as e.g. a store in the `unlock()` operation of a TAS spin lock (cf. Section 5.3.1). Please note that for simplification of the proposed implementation and the static WCET analysis it would be possible to prohibit the use of load/store accesses on synchronisation variables in coding guidelines for the programmer. Also, replacing the above mentioned store in TAS locks with a RMW operation is possible. However, in the following those load/store accesses on synchronisation variables are allowed, and are considered in the implementation and evaluation of the split-phase synchronisation technique. The split-phase synchronisation technique is integrated in the augmented memory controller, that is between the memory interconnect–the real-time bus–and the SDRAM (see Figure 5.2). The SDRAM logic is retained as standard SDRAM and is not subject to any changes by the split-phase technique. Without the split-phase synchronisation technique, the *reorder_buffer* and the *sync_buffer* would not be needed, but the *mem_buffer* and synchronisation logic is already included in the augmented memory controller (see Section 3.2). Despite the implementation in the memory controller of an SDRAM, it would also be possible to include the techniques in the controller of a shared (data) cache memory, e.g. an L2 cache, as long as predictability can be assured.

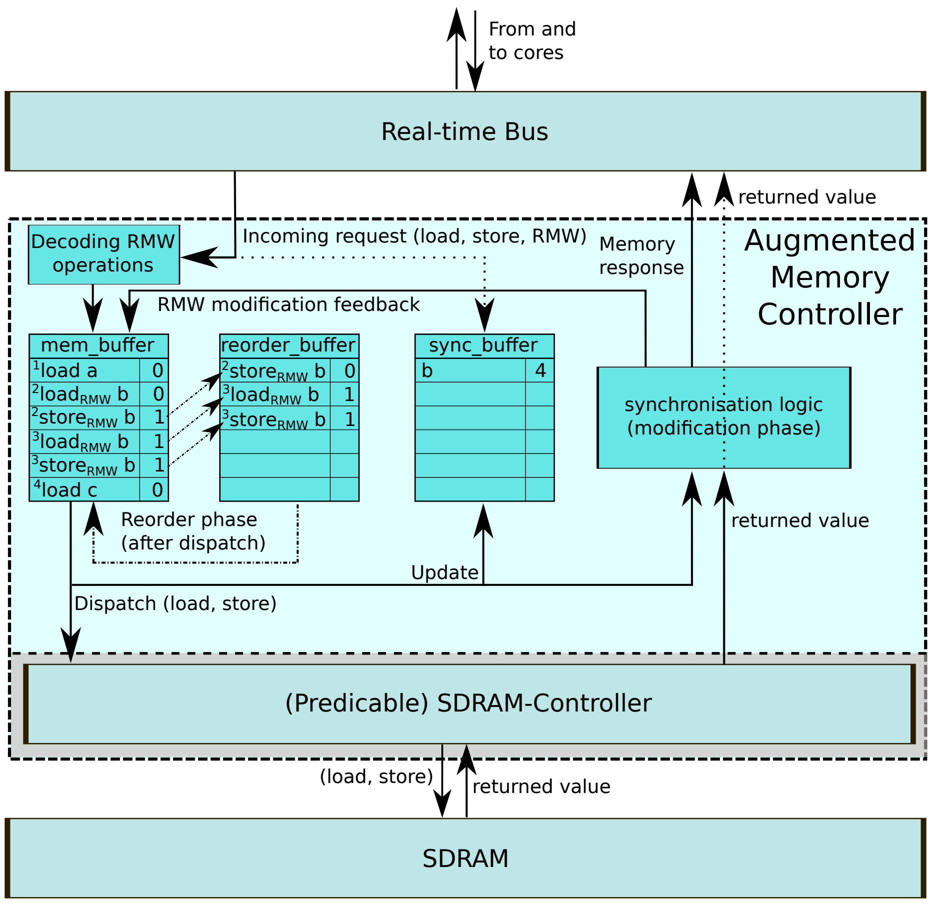


FIGURE 5.2.: Schematic overview of the augmented memory controller with additional implemented hardware for the split-phase synchronisation technique (see also Figure 3.3).

Memory requests are distinguished between load/store and RMW operations in the augmented memory controller. In Figure 5.2 the following syntax for different memory requests is used: $^{1}load\ a$ for a load from core 1 on memory address a, $^{2}load_{\mathrm{RMW}}\ b$ and $^{2}store_{\mathrm{RMW}}\ b$ for the load respectively the store phase of a RMW operations on memory address b from core 2. In the *mem_buffer*, respectively the *reorder_buffer*, each row represents a memory operation (in the first column) associated with a *reorder flag* (second column). The reorder flag is either '0' to flag a memory operation that should never be reordered (e.g. for normal load and store operations that do not access synchronisation variables), whereas a '1' as reorder flag allows reordering for the corresponding memory operation.

In the following sections, the course of events for memory operations with the split-phase technique are described, that is the handling of incoming requests, the dispatching of memory requests to the SDRAM logic, and the reordering of memory requests in case the split-phase synchronisation technique is applied to RMW operations.

5.2.1. Incoming Requests

For an incoming load/store operation the augmented memory controller first checks if the memory operation accesses a synchronisation variable that is already being accessed and therefore would be in the *sync_buffer* (see Figure 5.2). If not, the load/store operation is just added to the *mem_buffer* without setting the *reorder flag*. In the other case, it is added with the *reorder flag* set, and the counter of the accessed synchronisation variable is incremented in the *sync_buffer* (second column in each row of the *sync_buffer*). The *reorder flag* is used in the reorder phase (see Section 5.2.3), and memory accesses with the *reorder flag* set are not allowed to be brought forward as that could possibly breach data consistency. For example, if a store operation at memory address a arrives after a previous arrived RMW operation on the same memory address a, the store operation must be dispatched either *before* the RMW operation, or *after* it, but not in between.

Please note that if normal load or store operations on synchronisation variables are forbidden, e.g. by coding guidelines (see also discussions in Sections 3.3 and 4.3), it would be possible to simplify the above presented process. Then, in the case only RMW operations are allowed to access synchronisation variables, the other load/store operations do not need to check whether they access a synchronisation variable, and thus are just appended to the *mem_buffer* without the reorder flag set. This could be useful if the comparison of accessed memory addresses with the content of the *sync_buffer* takes too long, which mostly depends on the size of the *sync_buffer* and the frequency at which the memory controller operates. Then, that additional latency would only apply to RMW operations, but does not influence normal load and store operations. However, then it must be assured that memory operations on synchronisation variables are always and only done with RMW operations, as otherwise data inconsistencies are possible.

When a RMW operation is detected, the load and store accesses are split, and, if no other synchronisation request on that variable is stored in the *sync_buffer*, the memory address of the RMW operation is added to the *sync_buffer* with the counter set to two. In the *mem_buffer* both accesses are stored, where only the *reorder flag* of the $store_{\mathrm{RMW}}$ is set, but not for the $load_{\mathrm{RMW}}$ access. On the other hand, if there is already an access to that synchronisation variable in the *sync_buffer*, the counter for that address will be increased by two (e.g. to four as depicted in Figure 5.2 for the synchronisation variable b), and both split accesses are stored in the *mem_buffer* with the *reorder flag* set. This is done as in the reordering phase it must be assured that this RMW operation must not start before the store phase of the previous ongoing RMW operation on the same memory address is completed to maintain data consistency. Nonetheless, memory operations that are not accessing the same synchronisation variable can be executed in between the latter RMW operation, or even before it (details are presented in Section 5.2.3).

The size of the *sync_buffer* must be sufficient to hold N entries, where N is the maximum number of concurrently executed hardware threads in the multi-core processor. For the case of the MERASA multi-core processor, which only allows one HRT thread to be executed on each core, it is sufficient to reserve one row for each core in the *sync_buffer*. However, in a more general case, the number of rows needed in the *sync_buffer* corresponds with the number of possible concurrently executed threads.

5.2.2. Dispatching

Each time the memory controller is ready to dispatch a new request from the *mem_buffer*, it checks the *reorder flag* of the first entry in the *mem_buffer*. If the *reorder flag* is not set, that memory request is dispatched. Else, the next memory request without the *reorder flag* set will be selected from the *mem_buffer* and dispatched, and the reordering starts. If there is no request without the flag set, the first entry is dispatched and also the reordering phase starts.

An example in which only memory operations with the reorder flag set are in the *mem_buffer* can be constructed from Figure 5.2. If the memory access $^4load\ c$ would not be in the *mem_buffer*, and no new memory accesses enter the memory controller, only memory operations with the reorder flag set would be in the *mem_buffer* after the first two accesses ($^1load\ a$ and $^2load_{\mathrm{RMW}}\ b$) are dispatched. Then, the next memory operation dispatched would be $^2store_{\mathrm{RMW}}\ b$, even with the reorder flag set, as it will be the first in the list, and the other two remaining memory operations ($^3load_{\mathrm{RMW}}\ b$ and $^3store_{\mathrm{RMW}}\ b$) also have the reorder flag set.

When a memory access to a synchronisation variable is dispatched, the counter of the corresponding memory address in the *sync_buffer* is decremented (see edge labelled *Update* from *mem_buffer* to *sync_buffer* in Figure 5.2). Furthermore, the *synchronisation logic* is notified what kind of memory access is currently processed. This is needed as either the memory access will be finished and dispatched directly to the cores over the real-time bus (dotted edge in Figure 5.2), e.g. for a normal load or store operations. Or, for a RMW operation that does not need the loaded value for modification, that is a TAS operation, the synchronisation logic updates the corresponding store in the *mem_buffer* without losing an extra cycle for the modification in the synchronisation logic. All other RMW operations, for instance an F&I/F&D operation, need to modify the loaded value and then transfer it to the corresponding store of that RMW operation in the *mem_buffer*. Dispatched memory accesses are standard SDRAM accesses, and are handled in the same way as without the split-phase synchronisation technique (also see Section 5.2.5).

5.2.3. Reordering

The reordering phase is the main phase of the split-phase synchronisation technique. It assures the best possible worst-case for load and store operations, that is for N cores only load/store operations of the $N-1$ other cores are executed before each load/store operation. Also, the reordering ensures that the load and store phase of RMW operations are not deferred infinitely and are eventually dispatched with the best possible WCML that can be achieved under the requirement that load/store operations are handled with priority. The reordering phase kicks in if the memory access on top of the *mem_buffer* has the reorder flag set. However, this means that that access is not executed as long as other memory requests without the reorder flag set are in the *mem_buffer*. In Section 5.3 details on the worst-case access behaviour and the corresponding WCMLs for normal memory accesses and those on synchronisation variables are presented.

In the reordering phase all accesses in the *mem_buffer* with the *reorder flag* set are moved to the *reorder_buffer*. For the first access that is moved to the *reorder_buffer*, e.g. ${}^{2}store_{\mathrm{RMW}}\ b$ in Figure 5.2, the *reorder flag* is set to '0'. It is instantly dispatched if no other memory request without reorder flag, that is a normal load/store operation, is waiting in the *mem_buffer*. The deletion of the reorder flag is needed, as otherwise the waiting load/store phase of a RMW operation might be deferred infinitely by incoming concurrent normal load/stores of other cores that would be executed before that waiting store (see also the worst-case access pattern in Figure 5.3). When all accesses in the *mem_buffer* are processed, the accesses in the *reorder_buffer* are appended, with the updated reorder flags, to the *mem_buffer*.

For instance, in Figure 5.2 the reorder phase would start after the two accesses ${}^{1}load\ a$ and ${}^{2}load_{\mathrm{RMW}}\ b$ are finished, and the first access on top of the *mem_buffer* would be an access (${}^{2}store_{\mathrm{RMW}}\ b$) with the reorder flag set. Then, the accesses with the reorder flag set are moved to the *reorder_buffer*, and the ${}^{4}load\ c$ access would advance to the top of the *mem_buffer*. The reorder flag of the ${}^{2}store_{\mathrm{RMW}}\ b$ access, that is the first memory operation in the *reorder_buffer*, is deleted (as depicted in Figure 5.2), and then the three accesses in the *reorder_buffer* are moved and enqueued in the *mem_buffer*. The implication of reordering on data consistency is discussed in detail in Section 5.2.4.

5.2.4. Consistency and Atomicity of RMW Operations

A mandatory requirement is to maintain atomicity and consistency of RMW operations, meaning that the parallel program must be still functional correct when using the split-phase synchronisation technique.

Atomicity

Atomicity of split RMW operations is trivially satisfied. That means that 1) neither the accessed variable is changed by other accesses than the ongoing RMW operation, and 2) nor can the RMW operation finish incomplete (e.g. meaning that the store phase never finishes). Load/store accesses to other variables are brought forward and executed in between the load/modification phase and the store phase of a split RMW operation. However, this means that also load or store phases of a split RMW operation could access other variables than an ongoing other RMW operation, and therefore those load or store accesses could also be executed in between or before the other RMW operation. Though, the accessed variable is not changed between the load/modification phase and the store phase, and requirement 1) holds. Through the logic in the reordering phase it is achieved that every waiting memory request is dispatched eventually, that is the waiting time for every access to be finished has an upper bound. So, 2) is also satisfied, and thus the split-phase synchronisation technique does not breach the atomicity of RMW operations. The atomicity of normal load/store operations is trivially asserted in the MERASA multi-core processor as loads/stores only access one memory word at a time. Please note that for processors that read or write in bursts (e.g. to fetch a complete cache line), this might not be the case.

Consistency

It is assumed that the programmer takes care of explicit synchronisation, e.g. critical sections are secured with locks and temporal dependencies are handled with barriers—both implemented with RMW operations as detailed in Chapter 3. Also, it is presumed that the hardware and software implement consistency as described in Section 2.2, that is the weak consistency model–*weak ordering*–as proposed by Adve and Hill (1990). They define *weak ordering* "[i]n a multiprocessor system" as follows: "[...] storage accesses are weakly ordered if (1) accesses to global synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed, and if (3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed."

It must be assured that the split-phase synchronisation technique maintains these requirements to adhere to the *weak ordering* consistency model. The MERASA multi-core processor, due to in-order program execution in the cores, only dispatches one memory request from a core at a time to the memory controller. Shared data is only stored in one memory location, the global memory, so memory operations on shared data are complete (and hence visible) for all cores after the memory operation finishes. Also, as stated above, critical sections need to be secured by the programmer, e.g. with locks as described in Chapter 3: in Section 3.1.3 consistency has been discussed for synchronisation operations without the split-phase synchronisation technique. Access to shared synchronisation variables is done with RMW operations, which are implemented atomically. The split-phase synchronisation technique splits a RMW operation into two memory operations, a load and a store operation, and allows to execute other memory operations in between.

The requirement (1), strongly ordered accesses to synchronisation variables, is maintained by the use of *reorder flags* in the augmented memory controller and atomicity of RMW operations. For each incoming memory access to a synchronisation variable, either a load, store, or RMW operation, an entry in the *sync_buffer* is added or updated. According to already ongoing or waiting memory requests on synchronisation variables, *reorder flags* of incoming memory accesses are assigned (see Section 5.2.1). Together with the FIFO logic of the *mem_buffer* and *reorder_buffer*, the logic of the *reordering phase*, and atomicity of memory operations as detailed above, it is assured that the access to synchronisation variables is strongly ordered by a *first come, first serve* (FCFS) policy. Though, the FCFS policy only holds from the view of the memory controller, as depending on the arbitration of memory requests in the interconnect—in the case of the MERASA multi-core processor this is a round-robin arbitration between cores at the bus interconnect—memory requests might not be answered in the order they are ready at the core level, but in the order they arrive at the memory controller. Nonetheless, to maintain *weak ordering* consistency, it is sufficient that accesses to shared synchronisation variables are strongly ordered in the memory controller, as the "storage access" (see Adve and Hill 1990) is executed directly after the arbitration in the memory controller, and thus requirement (1) is satisfied.

In the following it is shown that *weak consistency* requirements (2) and (3) are also retained with the split-phase synchronisation technique. As an example a parallel program part with *Pthreads* from POSIX (2008) is used in which each thread/core executes the same *worker* function with concurrent accesses to a lock-secured critical section (see Listing 5.1), respectively a barrier (see Listing 5.2).

Listing 5.1:
Example parallel code parts for critical sections secured by locks for discussions on consistency with the split-phase synchronisation technique.

```
Core 1                                   Core 2

A1) pre-critical section code            B1) pre-critical section code

A2) lock entry code                      B2) lock entry code
A3) critical section                     B3) critical section
A4) lock exit code                       B4) lock exit code

A5) post-critical section code           B5) post-critical section code
```

Listing 5.2:
Example parallel code parts for barriers for discussions on consistency with the split-phase synchronisation technique.

```
Core 1                                   Core 2

C1) pre-barrier code                     D1) pre-barrier code

C2) barrier entry code                   D2) barrier entry code
C3) waiting                              B8) waiting
C4) barrier exit code                    B9) barrier exit code

C5) post-barrier code                    D5) post-barrier code
```

With the help of the example in Listings 5.1 and 5.2 it is shown, that neither global shared data accesses, e.g. from the critical sections (see lines A3/B3) in Listing 5.1), conflict with accesses on global shared synchronisation variables (requirement (2) in the *weak consistency* model), that is memory operations on synchronisation variables in the lock entry/exit code (see lines A2/B2 and lines A4/B4 in Listing 5.1), nor the other way around (requirement (3) in the *weak consistency* model). The same must hold for the code sections C1/D1 to C5/D5 for barriers in Listing 5.2.

As a reminder, it is mandatory that an access to global shared data is explicitly secured with locks by the programmer, which is part of the demands of the *weak consistency* model. If, for instance, in the example in Listing 5.1, the programmer would not secure the access to shared data accessed in lines A3/B3 within a critical section, e.g. lines A2 and A4 would not be existent, requirement (2) and (3) of the *weak consistency* model could fail.

That is the lines A3 and B2/B3/B4 are then executed concurrently by *Core 1* and *Core 2* without the required mutual exclusion. Please note that this would even conflict with the weak consistency model when no reordering is performed by the split-phase synchronisation technique.

Requirement (3) of the *weak consistency* model prohibits to access shared global data before correct access synchronisation has been performed, e.g. in the example in Listing 5.1 the access to memory operations in line A3 is only allowed after the synchronisation operation in line A2 is successfully finished. Vice versa, requirement (2) of the *weak consistency* model implies that synchronisation operations are only started after previous memory operations on global shared data are finished. For instance, the synchronisation operations in line A4 that unlock the critical section need to be executed after all memory operations from line A3 are finished. This is trivially true for one core, as memory operations issued from a core are ordered, and only one memory operation is dispatched at a time: no other memory operation from that core is dispatched until the previous is finished (cf. Section 4.1.3). Thus, also no memory operation from line A1 are executed after or in between memory operations A3, as well as no memory operations from line A5 are executed before or in between memory operation from line A3, respectively A1.

Moreover, if *Core 1* executes a memory operation from line A3, that is in the critical section, *Core 2* must not execute memory operations from line B3 as well to not break the consistency requirement. *Core 1* only executes memory operations from line A3, if and only if the previous synchronisation in the lock entry code in line A2 are completely and successfully finished, that is *Core 1* holds the lock for the critical section (lines A3/B3). To retain the requirements of the *weak consistency* model, *Core 1* must not execute operations from line A3, as long *Core 2* has not finished the lock exit code in B4. This holds, as *Core 1* can only progress from line A2 to A3 if the lock is successfully gained, and that is only true after the lock has been successfully released by *Core 2*, so, all synchronisation operations in line B4 are finished. This also holds with the split-phase synchronisation technique as accesses to the same synchronisation variable are strongly ordered as presented above. Therefore, when *Core 1* is in the critical section (line A3), *Core 2* has either not acquired the lock and is executing memory operation from line B2 or before, or *Core 2* already released the lock for the critical section and therefore executes line B5.

The very same holds for the barrier example beginning in lines C1/D1, as possibly accessed global data in the pre-barrier code in lines C1/D1 must be executed *before* accesses in the post-barrier code in lines C5/D5. This holds with the split-phase synchronisation technique, as only one memory request per core is processed in the memory at a time, so no memory requests from lines C5, respectively D5, can execute before memory requests from line C1, and accordingly D1. In addition, no memory requests from D5 are executed before memory requests from C1, and correspondingly for memory requests from C5 and D1, as this could only happen if *Core 2*, respectively *Core 1*, would both execute memory requests from lines C2/D2 to C4/D4, which is, in the same case as above, not possible, as only one memory request per core is handled at a time (also see discussion on software barriers in Section 3.3.6).

5.2.5. Related Work

The term *split-phase synchronisation* technique, introduced in this chapter, is not related to the commonly known *split-phase access* introduced by Culler et al. (1993) in Split-C.

The split-phase synchronisation technique uses a similar technique as the LL/SC primitive (also see Sections 2.2.4 and 3.2.4), which is e.g. used in the Alpha AXP (Sites 1993), PowerPC (PowerPC ISA 2010), ARM (ARMv6-M ISA 2010, ARMv7-M ISA 2010), and MIPS (MIPS32 ISA 2003) ISAs. The LL/SC implementations apply a coarse-grained approach, namely they do not monitor changes on the granularity of memory words, but lines of memory or even complete memory pages. LL/SC was initially intended to scale well on large multiprocessors with distant shared memory. In summary, LL/SC is a RMW operation implemented as part of the ISA, whereas the split-phase synchronisation is a technique used on all implemented RMW operations inside the memory controller. It splits their load, modification and store phases to reduce the worst-case memory latencies of loads/stores by prioritising them over concurrent RMW operations, and uses a fine-grained approach monitoring accessed synchronisation variables in the memory controller.

Split-phase memory operations are also known from dataflow computing. The so-called I-structures (see Arvind et al. 1989, Culler et al. 1991, Ungerer 1993) for the access on big data structures in dataflow computers use read operations (I-Fetch) with split phases, that is the actual read request is queued if the data element is not available yet (marked with an *empty* bit). The process, however, can meanwhile progress, and the requested value from the *I-Fetch* is returned as soon at is available (see also Section 3.2.4).

Akesson and Goossens (2011) introduce a pre-emptive service enabled by an *atomizer* to maintain robustness in a predictable memory controller implementation using a static-priority scheduler. This technique is similar to the split-phase synchronisation technique. Though, the main goal of the split-phase synchronisation is to reduce the pessimism and overestimation in the WCET analysis arising from the loss of information concerning the concurrent accesses of cores at shared resources (see Cullmann et al. 2010), e.g. the global memory.

Monchiero et al. (2005) present an augmented global memory controller, the Synchronisation-operation Buffer (SB), to reduce contention for busy-waiting synchronisation primitives in future mobile systems with complex NoCs. Their focus is on reducing contention, and therefore enabling an efficient use of busy-waiting synchronisations like spin locks. The goal of their technique is to decrease the average-case execution time by speeding up slow synchronisation primitives, while also enabling fine-grained synchronisations. Opposing to that, the focus of the augmented memory controller and the split-phase synchronisation technique introduced in this thesis is on speeding up frequent memory operations, like loads and stores, while slower synchronisation primitives, that is RMW operations, are potentially delayed, resulting in an improved overall worst-case performance, that is a lower worst-case guarantee. In conclusion, timing predictability is highly influenced by the timing variability of different memory response times rather than it is impacted by contention with the static timing analysis applied in this thesis.

A likewise approach of a synchronisation buffer technique, the Request-Store-Forward (RSF) model, is proposed by Liu and Gaudiot (2007). Their RSF synchronisation model splits synchronisations in three phases. In the request phase a process sends a synchronisation request to the synchronisation buffer. Then, in the second phase, the request is stored in a FIFO queue if the request cannot be served immediately while also switching the requesting process into sleep mode to eliminate contention. In the third phase, the forward phase, the synchronisation buffer notifies a waiting process from the waiting queue and sends the requested data, after the previous synchronisation request is finished. The RSF model is, similar to I-structures and M-structures introduced in dataflow computing (cf. Section 3.1.2), a more coarse-grained approach of splitting synchronisation accesses compared to the fine-grained splitting of RMW operations with the split-phase synchronisation technique proposed in this thesis.

Zhu et al. (2007) introduces a further synchronisation buffer, the Synchronisation State Buffer (SSB), to achieve efficient fine-grained synchronisations in many-core processors. Also, the aim of the SSB is to achieve better average-case performance for high performance many-core processors. The SSB resides at each memory bank and caches memory locations that are currently accessed by SSB synchronisation operations.

Though, the focus of the above techniques with synchronisation buffers is to decrease the average-case execution time by speeding up slow synchronisation primitives, while also enabling a fine-grained synchronisation. Furthermore, I-Structures and the RSF model additionally aim to eliminate contention by suspending the waiting processes. Contrarily, the split-phase synchronisation technique works on a even more fine-grained level by splitting atomic RMW operations in the augmented memory controller. The split-phase synchronisation technique aims at reducing the impact of pessimism in static timing analyses introduced from the interference of slow memory operations with faster ones. Hence, it supports WCET-efficient HRT capable synchronisation primitives, while also reducing the pessimism and overestimation in static WCET analyses (see also Section 4.1.2).

Although memory buffers, e.g. for store operations, are not implemented in the MERASA multi-core processor employed in this thesis, the split-phase synchronisation technique reorders memory accesses based on their nature, that is normal load/store accesses are prioritised over RMW memory operations. The reordering in the augmented memory controller meets the demands of a weak consistency and data-race-free memory model (see Adve and Hill 1990, cf. Section 5.2.4), as detailed in Section 5.2.4. To further ease writing parallel programs, Devietti et al. (2011) propose a relaxed consistency deterministic computer (RCDC) based on deterministic shared-memory multiprocessing (DMP) (see Devietti et al. 2010). Their approach (DMP-HB) uses the data-race-free relaxed memory model (see Adve and Hill 1990) with making use of *happens-before* (HB) relations to achieve a lower amount of required memory fences. It might be interesting to explore how such techniques can be implemented in the augmented memory controller proposed in this thesis (see Section 3.2) in concert with the split-phase synchronisation technique, and also to investigate the impact of the DMP-HB algorithm on timing predictability in embedded shared-memory multi-core processors in general.

5.3. Split-Phase Synchronisation Technique Evaluations

In the following the WCMLs of memory operations with the split-phase synchronisation technique are formally determined, and WCET guarantees of two parallelised HRT programs are compared for their execution with different synchronisation techniques and with and without the split-phase synchronisation technique. The evaluation settings for the following sections are the same as introduced in Section 4.1.3.

5.3.1. WCMLs with the Split-Phase Synchronisation Technique

Two cases to determine the WCMLs of a HRT thread's memory requests with the split-phase synchronisation technique can be distinguished: 1) load/store operations on non-synchronisation variables, and 2a) RMW operations respectively 2b) load/store operations on synchronisation variables.

By prioritising load/store operations in the augmented memory controller with the split-phase synchronisation technique, the WCML of a load/store from Equation 4.2 decreases, that is the load/store operation has to wait for the NHRT memory request of its own core, and load/store operations of other cores, but not for RMW operations of other cores. As a load operation T_L takes longer than a store operation, it has to be assumed that, in the worst-case, the other cores issue load operations, respectively a load phase of a split RMW operations. Therefore, for case 1), the WCML for a load/store on non-synchronisation variables is calculated rather simply as follows:

$$T_\mathrm{WCML} = T_\mathrm{HRT} + 2 \cdot T_\mathrm{B} + T_\mathrm{max} + (N-1) \cdot T_\mathrm{L} \tag{5.1}$$

For the cases 2a) and 2b) the worst-case scenario is more complex. Figure 5.3 depicts that worst-case scenario for case 2a), however, it also shows the case 2b) that, in the worst-case, finishes in cycle 60 (59) for a load (store) on a synchronisation variable. To explain that worst-case scenario in detail, an ordered list of operations σ is introduced, where ${}_{L}\sigma_x$ is a load operation of core x, and ${}_{LP}\sigma_y$ and ${}_{SP}\sigma_y$ are the load phase and store phase of a RMW operations of core y. ${}_{SP}\sigma^*_y$ stands for the store phase of a RMW operation of core y with the reorder flag set (see Section 5.2.3), that is an operation with lower priority. Now, it is necessary to keep in mind that in the reorder phase of the split-phase synchronisation technique an operation ${}_{SP}\sigma^*_y$ transforms into ${}_{SP}\sigma_y$ when the reorder flag is deleted. With the consistency requirement in the MERASA processor that only one memory operation of a core can be active at a time, and with N cores, an ordered list of memory operations ${}_{L}\sigma_2 >_L \sigma_3 > \ldots >_L \sigma_N >_{LP} \sigma_1 >_{SP} \sigma^*_1$ in the memory controller is achieved, with ${}_{L}\sigma_2 >_L \sigma_3$ meaning that ${}_{L}\sigma_2$ is executed before ${}_{L}\sigma_3$.

For the worst-case scenarios in Section 4.2 without the split-phase synchronisation technique that ordered list was never changed, as no σ^* operations were involved, that is memory operations with reorder flag set. Therefore, the worst-case was rather simple to compute. For the cases 2a) and 2b) σ^* operations need to be covered. The worst-case scenario for a memory operation of core 1 is then after cycle 14 in Figure 5.3 as follows: ${}_{LP}\sigma_2 >_{SP} \sigma^*_2 > \ldots >_{LP} \sigma^*_N >_{SP} \sigma^*_N >_{LP} \sigma^*_1 >_{SP} \sigma^*_1$.

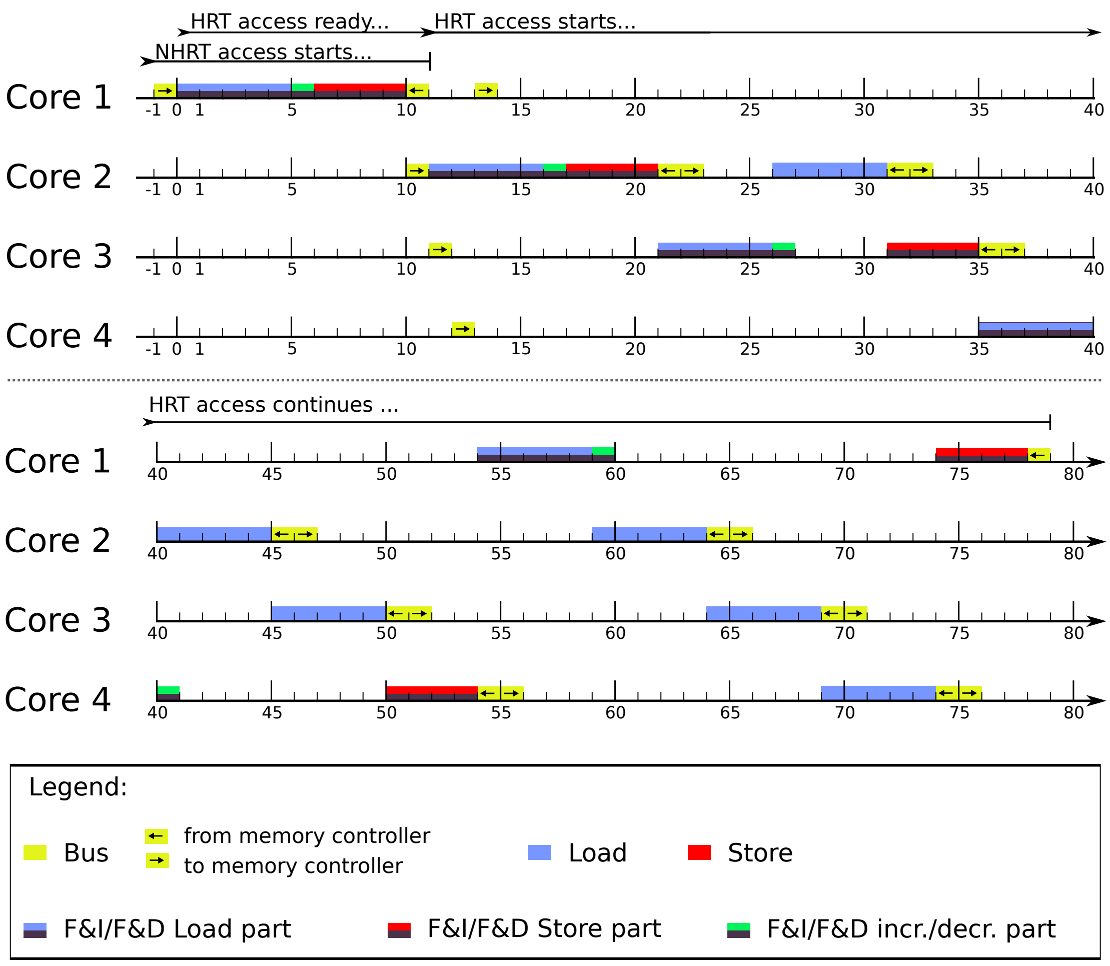


FIGURE 5.3.: WCMLs for a HRT RMW operation of *Core 1* in a quad-core processor with the split-phase synchronisation technique.

Now, it needs to be assumed that once one of the other cores finishes its memory operation, it sends a new memory request. To represent the worst-case, these memory operations need to be σ operations (e.g. the loads of core 2 in cycles 26, 40, 59), as then these memory operations are executed before the σ_1 operations (see cycles 54 and 74 of core 1 in Figure 5.3). For σ^* operations of the other cores this would not hold, as they would be executed after the σ^* operations of core 1, and therefore not representing the worst-case. Taking this into account, in the worst-case $\sum_{i=1}^{N-1} i$ operations ${}_L\sigma$ are executed before the ${}_{SP}\sigma_1^*$ operation. In summary, for an N-core processor with $N > 2$, the WCML can then be computed as:

$$T_{\text{WCML}} = 2 \cdot T_{\text{B}} + (N+1) \cdot T_{\max} + \frac{N \cdot (N-1)}{2} \cdot T_{\text{L}} - (N-1) \qquad (5.2)$$

For case 2b), as mentioned above, the WCML of loads/stores on synchronisation variables is similar to the WCML of RMW operations. But, the store and modification phase is omitted. An access to a synchronisation variable then starts, in the worst-case, in the same cycle as the load phase of a RMW operation in Core 1 as depicted in Figure 5.3, but finishes already in cycle 59 (store) respectively in cycle 60 (load). The WCML is then calculated for $N > 2$ as:

$$T_{\text{WCML}} = T_{\text{HRT}} + 2 \cdot T_{\text{B}} + N \cdot T_{\max} + (N-1) \cdot T_{\text{L}} - (N-2) \tag{5.3}$$

In Table 5.1 the WCMLs of memory accesses on a quad-core respectively an eight-core MERASA processor with and without the split-phase synchronisation technique are depicted. For both configurations, the memory latency of a load operation has been fixed to either five cycles or ten cycles (for a store operation that means four respectively nine cycles latency). The WCMLs for normal loads/stores is decreased for every configuration, i.e. up to 40 % for an eight-core with ten cycles load latency (bottom right of Table 5.1). Other memory operations on synchronisation variables, that is load/store operations or RMW operations, have higher WCMLs with the split-phase synchronisation technique. The impact of the split-phase synchronisation technique on the pessimism in the WCETs, and under which circumstances the split-phase technique is proven to be beneficial in terms of WCET guarantees, is discussed below.

Table 5.1.:
Parametric WCMLs for different memory operations (Mem.Op.) in the MERASA WCET model for four, resp. eight cores, and a memory latency of five, resp. ten cycles with (denoted as WCML$^{\text{SPS}}$) and without the split-phase synchronisation technique.

# Cores	Memory Latency for a Load Operation					
	5 cycles			10 cycles		
	Mem.Op.	WCML	WCML$^{\text{SPS}}$	Mem.Op.	WCML	WCML$^{\text{SPS}}$
4 Cores	load	47	32	load	92	62
	store	46	31	store	91	61
	ld/st(sync)	47/46	60/59	ld/st(sync)	92/91	118/117
	TAS	51	79	TAS	101	159
	F&I/F&D	52	79	F&I/F&D	102	159
	5 cycles			10 cycles		
	Mem.Op.	WCML	WCML$^{\text{SPS}}$	Mem.Op.	WCML	WCML$^{\text{SPS}}$
8 Cores	load	87	52	load	172	102
	store	86	51	store	171	101
	ld/st(sync)	92/91	119/118	ld/st(sync)	172/171	234/233
	TAS	91	225	TAS	181	455
	F&I/F&D	92	225	F&I/F&D	182	455

Figures 5.4, 5.5, 5.6, and 5.7 depict the WCMLs of different configurations in more detail. The impact on the WCET of parallel programs can be derived from Table 5.3 (see Section 5.3.3): the augmented memory controller with the split-phase synchronisation technique improves the overall WCET guarantee by 1.03 to 1.30.

5.3.2. Impact on Pessimism in the WCET

One major impact on the pessimism in the WCET stems from the lack of knowledge on parallel accesses to shared resources in parallel programs. From Table 5.1 the correlation of types of memory accesses and the impact on the estimated WCET in a quad-core MERASA processor can be calculated. If n depicts the percentage of executed normal loads/stores, and m the percentage of executed RMW and load/store operations on synchronisation variables in the worst-case path of a parallelised HRT program, the split-phase synchronisation technique produces better *upper bounds* if:

$$\begin{aligned} 32 \cdot n + 79 \cdot m &\geq 47 \cdot n + 52 \cdot m \\ \Rightarrow 32 \cdot n + 79 \cdot (1-n) &\geq 47 \cdot n + 52 \cdot (1-n) \\ \Rightarrow n &\geq \frac{27}{42} \end{aligned} \tag{5.4}$$

with $n, m \in [0,1] \wedge n + m = 1$.

So, for four cores, the split-phase technique achieves lower *upper bounds*, if more than 64.3 % of all memory operations in the worst-case path of a parallel program are load/stores, or, in other words, if less than 35.7 % are RMW operations.

In the general case, the impact can be calculated from Equations 4.2 (WCMLs of loads/stores/RMWs without the split-phase synchronisation technique), 5.1 and 5.2 (WCMLs of loads/stores/RMWs with the split-phase synchronisation technique). Using again n for the percentage of normal loads, and m for the percentage of RMW operations, the following inequation (that is also depending on the number of cores N, and the latency T_{L} of a load operation) holds:

$$\begin{aligned} (T_{\mathrm{L}} + 2 \cdot T_{\mathrm{B}} + 2 \cdot N \cdot T_{\mathrm{L}}) \cdot n \quad &+ \quad (2 \cdot T_{\mathrm{L}} + 2 \cdot T_{\mathrm{B}} + 2 \cdot N \cdot T_{\mathrm{L}}) \cdot m \\ \geq (T_{\mathrm{L}} + 2 \cdot T_{\mathrm{B}} + (N-1) \cdot T_{\mathrm{L}}) \cdot n \quad &+ \quad (2 \cdot T_{\mathrm{B}} + (N+1) \cdot 2 \cdot T_{\mathrm{L}} + \frac{N \cdot (N-1)}{2} \cdot T_{\mathrm{L}} \\ &\quad -(N-1)) \cdot m \end{aligned}$$

with $T_{\max} = 2 \cdot T_{\mathrm{L}}$ and $n, m \in [0,1] \wedge n + m = 1$.

$$\begin{aligned} \Rightarrow (T_{\mathrm{L}} + 2 \cdot N \cdot T_{\mathrm{L}}) \cdot n \quad &+ \quad (2 \cdot T_{\mathrm{L}} + 2 \cdot N \cdot T_{\mathrm{L}}) \cdot m \\ \geq (T_{\mathrm{L}} + (N-1) \cdot T_{\mathrm{L}}) \cdot n \quad &+ \quad ((N+1) \cdot 2 \cdot T_{\mathrm{L}} + \frac{N \cdot (N-1)}{2} \cdot T_{\mathrm{L}} - (N-1)) \cdot m \end{aligned}$$

$$\Rightarrow N \cdot T_\mathrm{L} \cdot n \geq T_\mathrm{L} \cdot n + (\frac{N \cdot (N-1)}{2} \cdot T_\mathrm{L} - N + 1) \cdot m$$

$$\Rightarrow \frac{n}{m} \geq \frac{N}{2} - \frac{N+1}{(N-1) \cdot T_\mathrm{L}}$$

Now, if T_L increases, that is the latency of a normal load operation increases in the memory controller, the percentage of how many normal loads n need to be executed (so that the split-phase synchronisation technique is beneficial) converges. For simplification, the following assumption $T_\mathrm{L} \rightarrow \infty$, with $n + m = 1 \Leftrightarrow m = 1 - n$ leads to:

$$\frac{n}{1-n} \geq \frac{N}{2} \Rightarrow 2 \cdot n \geq N \cdot (1-n) \Rightarrow n \cdot (2+N) \geq N$$

$$\Rightarrow n \geq \frac{N}{N+2} \tag{5.5}$$

Solving Inequation 5.5 shows (that is if $T_\mathrm{L} \rightarrow \infty$): if at least $\frac{N}{N+2}$ of the executed memory operations in the worst-case path are loads/stores, the split-phase synchronisation technique produces lower *upper bounds*. Please note that this fraction is an upper bound on the percentage of needed executed normal memory operation in the worst-case path and would decrease if $T_\mathrm{L} < \infty$.

Table 5.2 presents the results of how many normal loads/stores are needed in the worst-case path for three to eight cores in the MERASA multi-core processor. Certainly, this may not hold for any number of cores, as e.g. the equation for the WCMLs of RMW operations on synchronisation variables includes the number of cores N as a quadratic term. However, eight cores connected over a shared bus to one memory controller is a feasible limit for a shared-memory multi-core processor (see Ungerer et al. 2010), and the split-phase technique is still beneficial if not more than 20 % of all memory operations in the worst-case path are RMW operations on synchronisation variables.

Table 5.2.:
WCET impact of the split-phase synchronisation technique for N cores in the MERASA multi-core processor. n depicts how many percentage of memory operations in the worst-case path need to be normal loads/stores to produce lower *upper bounds* in the WCET with the split-phase synchronisation technique.

Number of Cores N	Minimum Percentage n for (Normal) Loads/Stores
3	60.0 %
4	66.6 %
5	71.4 %
6	75.0 %
7	77.7 %
8	80.0 %

These results, however, only give a hint when looking at the source or binary code of a parallel program, as it denotes the correlation between executed memory operations in the worst-case path of the program. That is, to decide whether the split-phase synchronisation technique achieves better WCET guarantees, the really executed RMW operations versus the total memory operations in the worst-case path need to be taken into account. This is most likely different compared to a ratio that could be computed by just counting memory operations in the source/binary code, as loops and branches must be acknowledged. Still, comprising that parallel programs mostly contain only few synchronisation operations, e.g. many load operations are needed for instruction fetches, it can be concluded that the split-phase synchronisation technique is beneficial for the estimated WCETs of almost all parallelised programs. To quantify the different ratios, the OTAWA tool has been enhanced to output the number of RMW and total memory operations in the worst-case path. This allows for making practical decisions about the usefulness of the split-phase synchronisation technique. The evaluations in Section 5.3.3 show that the number of RMW operations in the worst-case path of those programs is between 2.82 % and 32.16 % for a quad-core MERASA processor. So, the requirement of more than 66.6 % of all memory operations in the worst-case path are normal loads/stores (see Table 5.2) holds for those examples.

The reduction of the WCMLs of a load operation with the split-phase synchronisation technique is depicted in the difference of Figures 5.4 and 5.5, whereas the increase of WCMLs for a RMW operation is shown by comparing Figures 5.6 and 5.7. The memory latency (x-axis in all figures) describes the number of cycles per load in the augmented memory controller. The number of cores (y-axis) is changed from four to eight cores, and the corresponding WCMLs (z-axis) are then presented as a 3D map. The WCMLs are computed on the basis of Equation 5.1 for the WCMLs of a load operation, and Equation 5.2 is used to compute the WCMLs of a RMW operation. The hue of grey colour in the figures depicts the WCMLs, that is a higher WCML results in a lighter grey, and a darker grey depicts a lower WCML. In the Figures 5.4, 5.5, 5.6, and 5.7 this grey scale is shown on the right hand side of each figure.

For the two different operations, that is load and RMW operations, the differences that are caused in the WCMLs when applying the split-phase technique versus not using the split-phase technique are easy to spot in the figures. For load operations in Figures 5.4 and 5.5, the WCMLs are much lower when using the split-phase synchronisation technique, and the impact of increasing the memory latency or the number of cores is not as high as for the case without the split-phase technique. However, for RMW operations, the results are different. As shown in Figures 5.6 and 5.7, the split-phase technique increases the WCMLs by a high factor, and as the number of cores is a quadratic term (see Equation 5.2), it does not scale well for a higher number of cores. Nonetheless, with and without the split-phase technique, the increase in the WCML of RMW operations is more sensitive to the number of cores than it is to the memory latency of a load operation (see also Section 4.2).

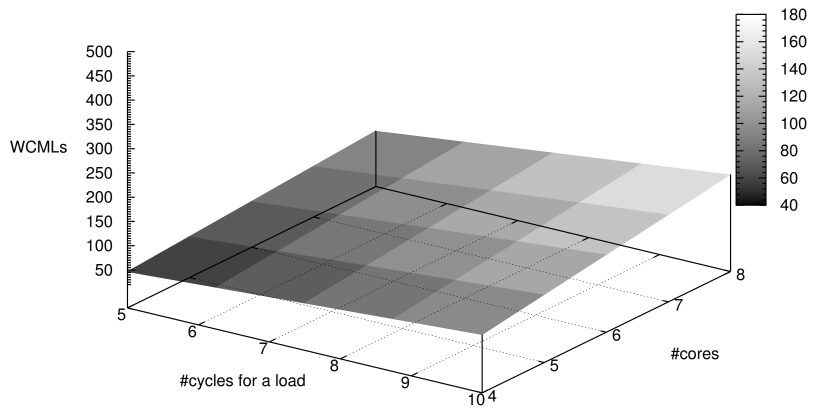


FIGURE 5.4.: WCMLs of load operations without the split-phase synchronisation technique represented in a 3D map. The hue of the grey squares represents the WCML for that configuration (see also the scale on the right side).

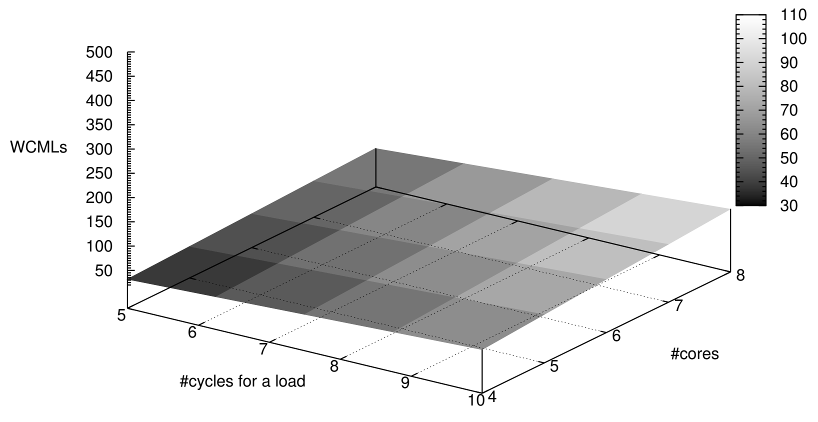


FIGURE 5.5.: WCMLs of load operations with the split-phase synchronisation technique represented in a 3D map. The hue of the grey squares represents the WCML for that configuration (see also the scale on the right side).

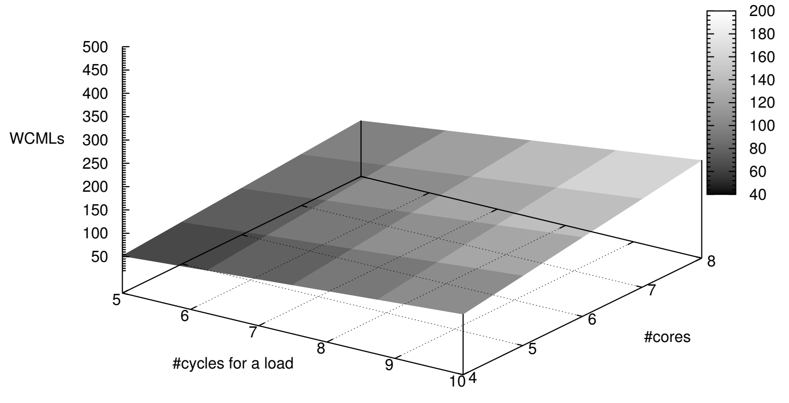


FIGURE 5.6.: WCMLs of RMW operations on synchronisation variables without the split-phase synchronisation technique represented in a 3D map. The hue of the grey squares represents the WCML for that configuration (see also the scale on the right side).

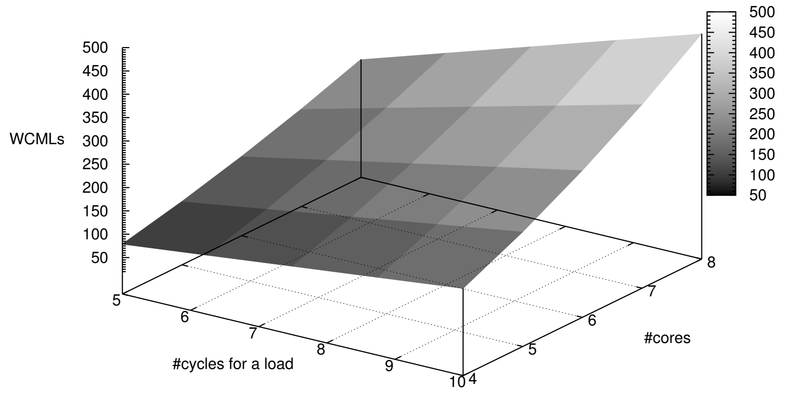


FIGURE 5.7.: WCMLs of RMW operations on synchronisation variables with the split-phase synchronisation technique represented in a 3D map. The hue of the grey squares represents the WCML for that configuration (see also the scale on the right side).

5.3.3. WCET Guarantees of Parallelised HRT Programs

In this section WCET guarantees of two parallelised programs, *matmul* and *IFFT*, are presented. Both have been implemented with three kinds of primitives to guard critical sections: mutex locks (see Section 3.3.3), binary blocking semaphores (see Section 3.3.5) and ticket locks (see Section 3.3.4). In addition, *IFFT* includes software barriers and was compiled with barriers implemented using either subbarriers and conditionals or the F&I primitive (see Section 3.3.6). Details on the two parallelised programs and WCET analyses without the split-phase synchronisation technique are presented in Section 4.4; estimated WCETs with the split-phase synchronisation technique are discussed below.

The WCET guarantees have been computed for a quad-core MERASA processor, with a bus latency of one cycle. In the WCET model no real SDRAM memory is simulated, but the upper bounded latency time for each memory access is used for the WCET estimates, that is five cycles for a load, and four cycles for a store. These values are typical for embedded SDRAM operating with up to 200 MHz and those values were derived with an in-house build FPGA prototype of the MERASA multi-core processor. For the synchronisation primitives nine (5+4) cycles for a TAS and ten (5+1+4) cycles for F&I/F&D are accounted for (also see Section 4.1.3). For a F&I/F&D operation the loaded value needs to be incremented or decremented, and the manipulated value is stored back. Therefore, the latency of a F&I/F&D is higher than for a TAS. In detail, one extra cycle needs to be taken into account in the memory controller to increment/decrement the loaded value. The WCWTs, introduced from the different software synchronisations, are included as detailed in Section 4.3.

TABLE 5.3.:
WCET estimates (# cycles) of synchronisation techniques for a quad-core MERASA processor from OTAWA with and without the split-phase synchronisation technique.

`matmul`	Mutex	Semaphore	Ticket Lock
without split-phase	1,589,483	1,221,477	1,044,762
split-phase(**WCET improv.**)	1,348,313(**1.18**)	1,058,139(**1.15**)	803,544(**1.30**)

`IFFT` Conditional Subbarriers	Mutex	Semaphore	Ticket Lock
without split-phase	288,781	224,196	204,933
split-phase (**WCET improv.**)	258,883 (**1.12**)	207,219 (**1.08**)	175,698 (**1.17**)

`IFFT` F&I Barriers	Mutex	Semaphore	Ticket Lock
without split-phase	209,572	135,420	120,817
split-phase (**WCET improv.**)	169,816 (**1.08**)	131,760 (**1.03**)	102,850 (**1.17**)

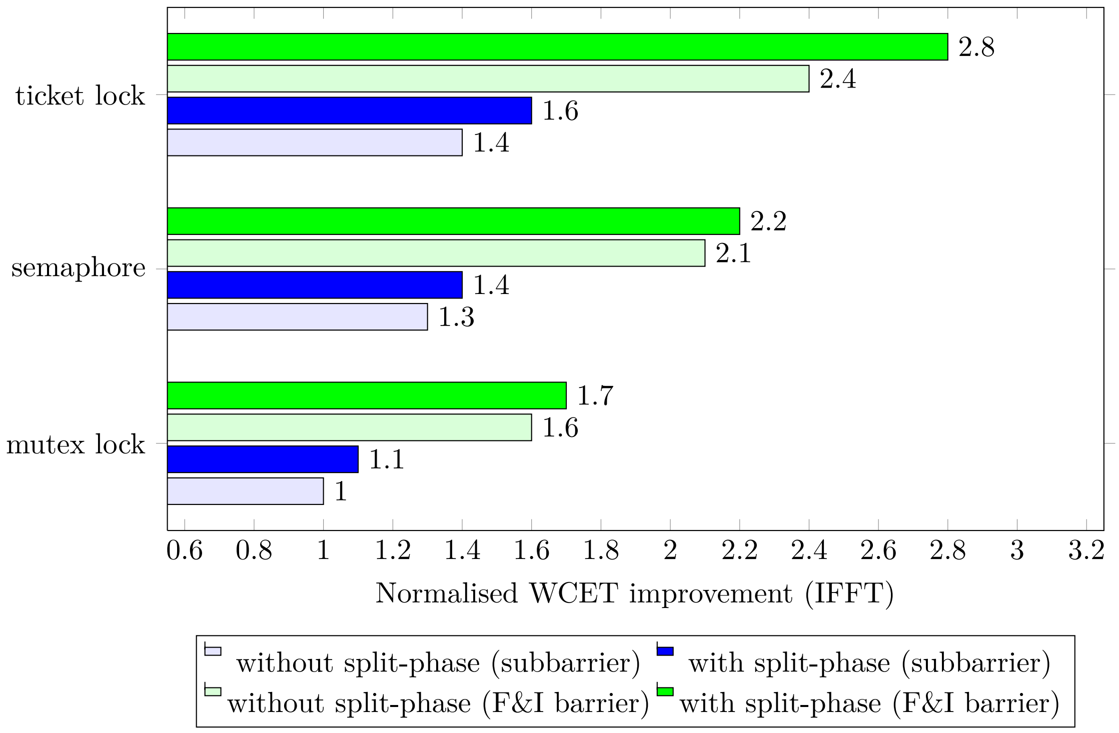


FIGURE 5.8.: WCET guarantee improvements on a quad-core MERASA processor for the parallelised *IFFT* using different software synchronisations with and without the split-phase synchronisation technique.

Figures 5.8 and 5.9 depict the improvement of WCET guarantees for the analysed four-threaded *IFFT* and *matmul* program. The WCET improvement is normalised on the reference WCET estimate derived from the parallelised *IFFT*, respectively *matmul*, with mutex locks, conditional subbarriers (only for *IFFT*; *matmul* uses no barriers), and without the split-phase synchronisation technique. On the one hand, F&I barriers outperform subbarriers, and ticket locks outperform binary semaphores and mutex locks, but the main point is the WCET improvement when using the split-phase synchronisation technique. The results in Table 5.3 show an improvement of the WCET guarantee when using the split-phase synchronisation technique of up to 1.17 for the *IFFT* program with ticket locks and both barrier implementations. From Table 5.3, the WCET improvement for *matmul* with the split-phase synchronisation is up to 1.30, that is for the *matmul* program with ticket locks. Also, the results in Table 5.3, and Figures 5.8 and 5.9 show that a WCET improvement with the split-phase technique is achieved for all used software synchronisations.

Figure 5.8 shows that applying different barrier implementations for *IFFT* yields in high changes in the normalised WCET improvement. The use of F&I barriers, instead of using conditional subbarriers, produces better WCET guarantees—with and without the split-phase synchronisation technique—no matter which locking technique is used.

Overall, when taking all software synchronisations into consideration, the improvement of the WCET guarantees using the split-phase synchronisation technique is up to 2.8 for the parallelised *IFFT* program, that is the *IFFT* version with conditional sub-barriers, mutex locks and without the split-phase synchronisation technique compared to the version with F&I barriers, ticket locks, and with the split-phase synchronisation technique used (cf. Figure 5.8).

For *matmul* (see Figure 5.9) a similar improvement of the WCET guarantees is reached. The baseline for the normalised WCET improvement is again the version of *matmul* with mutex locks and without the split-phase synchronisation technique. The impact of the split-phase synchronisation technique is higher for *matmul* than for *IFFT* for all three locking techniques (cf. Table 5.3), that is up to 1.3 for ticket locks with versus without the split-phase technique. The overall WCET improvement for *matmul* when using ticket locks and the split-phase technique of 2.0 is similar to the improvement of *IFFT*, when the different barrier implementation are not taken into account. That is the WCET improvement of *IFFT* using F&I barriers and ticket locks with the split-phase technique vs. mutex lock without the split-phase technique is also around 2.

To discuss the effect of the split-phase synchronisation technique on the improvement of worst-case guarantees, Table 5.4 depicts the ratios of synchronisation operations, that is RMW operations against total memory operations in the worst-case path of the two analysed parallel programs.

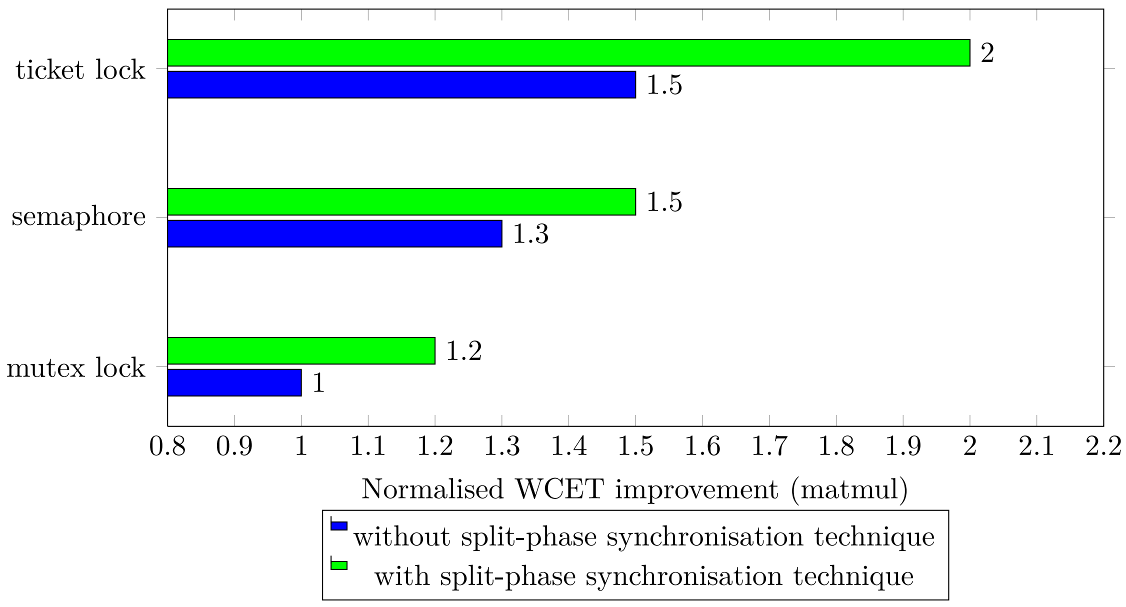


FIGURE 5.9.: WCET guarantee improvements on a quad-core MERASA processor for the parallelised matrix multiplication (*matmul*) using different software synchronisations with and without the split-phase synchronisation technique.

Table 5.4.:
Ratio of synchronisation operations vs. total memory operations in the worst-case path for *matmul* and *IFFT* derived from OTAWA. The special cases which show higher improvements in the WCET guarantees are in bold type (versions with ticket lock).

(a) `matmul` (30x30 matrix)

Synchronisation	Total Mem. Operations	Sync. Operations	Ratio
mutex lock	19,345	2,057	10.63 %
binary semaphore	16,258	2,732	16.80 %
ticket lock	16,710	472	**2.82 %**

(b) `IFFT` with subbarriers

Synchronisation	Total Mem. Operations	Sync. Operations	Ratio
mutex lock	1,835	279	15.20 %
binary semaphore	1,583	392	24.76 %
ticket lock	1,673	212	**12.67 %**

(c) `IFFT` with F&I barriers

Synchronisation	Total Mem. Operations	Sync. Operations	Ratio
mutex lock	1,280	204	15.94 %
binary semaphore	883	284	32.16 %
ticket lock	1,088	92	**8.46 %**

Table 5.3 shows that the improvement of worst-case guarantees with the split-phase synchronisation technique is higher for *matmul* as it is for *IFFT*, that is 1.30 for *matmul* versus 1.17 for *IFFT*. Also, the highest gain is achieved for the program configurations which apply the ticket lock synchronisation. The reason for this effect is the ratio of synchronisation memory operations to the total memory operations in the worst-case path of the parallelised programs, depicted in Table 5.4: for *matmul* this ratio is lower (between 2.82 % and 16.80 %) than for *IFFT* (between 8.46 % and 32.16 %). As already discussed above, the possible gain with the split-phase synchronisation technique is sensitive to the number of synchronisation operations in the worst-case path. The critical ratio for which a benefit can be achieved depends on the number of cores, and for the used settings in the evaluation above with four cores it is 66.6 %. From the ratios presented in Table 5.4, it is easy to see that for the *IFFT* program with subbarrier and binary semaphores this ratio is only just satisfied. Therefore, the gain for this configuration with the split-phase synchronisation technique is rather low, that is 1.03 (cf. Table 5.3). On the other hand, the configuration of *matmul* with ticket locks shows a very low ratio of 2.82 %, and hence the highest improvement of worst-case guarantees of 1.30 (see Table 5.3).

Please keep in mind that both analysed programs have been chosen to benchmark the WCET of synchronisation primitives, hence ratios of RMW versus total memory operations in the worst-case path of other, real-world parallel HRT programs might be actually lower. Thus, even higher improvements of WCET guarantees might then be achieved.

The past may not repeat itself, but it sure does rhyme. – Mark Twain

6 Application of Synchronisation Techniques

Writing parallel programs is still a difficult and error prone effort. This is true for functional correctness, however, in real-time embedded systems further challenges arise from the requirement to fulfil non-functional properties, i.e. predictable timing and fault tolerance. In this thesis, the focus is on timing properties of synchronisations in parallel programs, though the functional behaviour and correctness also influences the timing behaviour of parallel HRT programs. In the previous chapters the basics and requirements for synchronisation of parallel programs in real-time systems are introduced (cf. Chapters 2 and 3), implementations for timing predictable synchronisations in hardware and software and their static timing analyses are presented (cf. Chapters 3 and 4), and optimisations for better worst-case performance described (cf. Chapter 5).

Besides the challenges in static timing analyses introduced from hardware complexity, coping with software complexity needs to be taken into account as well. One observation—e.g. as stated below by Gebhard et al. (2011)—is that standard coding guidelines are not sufficient to enforce predictable timing (even more so for parallel HRT programs):

> "Our experience with static timing analysis of embedded software systems shows that the analysis complexity varies greatly. As discussed above, the software structure strongly influences the analyzability of the overall system. Existing coding guidelines, such as the MISRA-C standard, partially address tier-one challenges encountered during WCET analysis. However, solely adhering to these guidelines does not suffice to achieve worst-case execution time bounds with the best precision possible. We usually suggest to document the software system behaviour as early as possible—desirably during the software design phase—to tackle the tier-two WCET analysis challenges. Otherwise, achieving precise analysis results during the software development testing and validation phase might become a costly and time consuming process." (Gebhard et al. 2011)

Yet, it is still challenging for a programmer to produce timing predictable parallel code, as well as statically analysing the WCET of the parallel code on a multi-core processor is a complex task. One problem in static timing analyses stems from missing information on concurrent execution.

In this chapter an approach is presented to ease the effort of static WCET analysis, and timing analysable and predictable parallelisation. The approach compromises the transfer of information from the point of view of a programmer to the timing analysis (and vice versa), as well as only using *parallel design patterns* and *synchronisation idioms* to enforce robust, timing analysable parallel program code. The use of pre-defined, timing predictable *synchronisation idioms* helps, on the one hand, programmers in choosing the right synchronisation mechanisms for a given platform, and, on the other hand, also includes the necessity of providing annotations for the static timing analysing. In summary, applying synchronisation techniques that are timing analysable and well-known to the timing analyser, enables to fabricate timing predictable parallel HRT programs.

In Section 6.1 a short introduction to design patterns in the domain of general purpose computing is given. Section 6.2 details how *parallel design patterns* for HRT systems are envisioned, and especially links the hardware and software implementations for predictable synchronisation, presented in this thesis, in the *synchronisation idioms* layer of those design patterns. The chapter closes with related work on parallel design patterns and parallelisation of real-time programs in Section 6.3.

6.1. Design Patterns

Design patterns have been first introduced by Alexander et al. (1977), providing experience and knowledge on how to build places and estate that are “alive”. The novel approach of Alexander et al. (1977) involves the residents in the conceptional design. Later, the idea of design patterns has been adapted in software engineering with the change that not the *user* is in the centre point of design, but the programmer.

According to Baroni et al. (2003), design patterns were first introduced to the domain of software engineering by Beck and Cunningham (1987) for graphical user interfaces for the object-oriented programming language *Smalltalk*. The final advent of design patterns in programming has been achieved by the so-called *Gang of Four* (GoF) with their publication of “Design Patterns - Elements of Reusable Object-Oriented Software” (Gamma et al. 1995). Also, the yearly conference on *Pattern Languages of Programs (PLoP)* and a series of books (Coplien and Schmidt (1995), Vlissides et al. (1996), Martin et al. (1997), Harrison et al. (2000), and Manolescu et al. (2006)) emerging from these events, made design patterns very popular in the domain of software engineering. Later also followed a series of books on *POSA - Pattern-Oriented Software Architecture* by Buschmann et al. (1996), and among others, for concurrent programming in JAVA (Lea 2003), *Patterns in JAVA* (Grand 2002), and design patterns for real-time systems by Douglass (2006).

6.1.1. Programming with Design Patterns

In the domain of software engineering, design patterns provide widely approved, reusable solutions to recurring problems (cf. Baroni et al. 2003), and examples and descriptions how they could and should be used. Therefore, design patterns should reflect forces and motivations, and state clearly which problem context they affect. Design patterns are preferably independent of platforms, architecture, etc., hence abstract representations. Design patterns should also be mostly independent of programming languages; however, design patterns, for example in the scope of object-oriented programming language, might be a bit different for JAVA respectively Smalltalk, C# or C++. In contrast, idioms also solve recurring problems, but with a limited scope, as e.g. presented by Coplien (1992) for C++. Idioms are mostly more specific about underlying execution platforms, application domains, or programming languages. In this thesis, also two layers of abstractions (see Section 6.2) are introduced: the *parallel design patterns* as platform-independent solutions, and *synchronisation idioms* as a platform-dependent solution for timing predictable synchronisation techniques, e.g. provided by an OS or API for a given multi-core platform.

Classification Design patterns are usually integrated in a pattern catalogue, a pattern system, or even more sophisticated in a pattern language. A pattern catalogue typically just collects and lists a number of design patterns, but does not reflect the connections between those design patterns. Pattern systems can be seen as an extension to pattern catalogues, allowing for simplification of extending existing patterns, and showing relations between them. Also, pattern systems reflect contextual dependencies and classifications, e.g. on different abstraction levels. Pattern languages are more complex and "[...] they also initiate with their users to guide them through the solution spaces they span." (Buschmann et al. 2007, p. 290) Pattern languages are a "real" language with grammar, syntax and context. They integrate the software engineer and lead him through different pattern systems to compose complex systems. References to other (diverse) design patterns form a network of design patterns.

Anti-Pattern Another kind of design patterns are so-called *anti-patterns*. They highlight what a programmer or user must not do. Often coding guidelines (see e.g. Holzmann 2006) rely on such anti-patterns, however, (good) anti-patterns go one step further, that is they do not only point out why a solution is a bad solution, but they also point out forces and motivation for it, and reference better solutions or fitting design patterns. Anti-patterns often motivate their application (or not-application) with what sounds like a good idea in the first place, but turns out to be a bad idea later. An example for such a motivation for an anti-pattern is the famous *Brooks' law*: "Adding manpower to a late software project makes it later." (see Brooks (1975) and the updated and revised 20th anniversary edition by Brooks (1995)).

6.1.2. Parallel Design Patterns

In an interview with John Hennessy and David Patterson in 2006, Hennessy stated: "[...] when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced." (see Olukotun 2006) In this context, Asanovic et al. (2009) present an outlook on future programming for highly parallel systems. They state that they envision that the general-purpose parallel programmer will work on frameworks, which are itself based on one specific parallel design pattern. As example Asanovic et al. (2009) cite the very successful use of *Ruby on Rails* for sequential processors, which is a framework based on the *Model-View-Controller* design pattern.

Mostly, the approaches incorporating parallel design patterns also include ideas from the structured parallelisation approaches from Carriero and Gelernter (1989) and Foster (1995) (a detailed summary on those approaches is given by Ungerer 1997, p.87ff). A structured integration of parallelisation approaches into a design pattern language for (general-purpose) high-performance computing is done by Mattson et al. (2004) (based on previous work of Massingill et al. (1999, 2001a,b)).

Mattson et al. (2004) present their pattern language categorised into three abstraction layers[1]. A high-level layer lists *structural patterns* (e.g. *Map Reduce* and Process Control) and *computational patterns* (*Dense Linear Algebra* and *Sparse Linear Algebra* for example). A mid-level layer consists of *parallel algorithm strategy patterns* (e.g. *task parallelism* and *data parallelism* and *implementation strategy patterns*, which is further split into program and data structure. The program structure patterns include *master-worker* and *fork-join* patterns, while the data structure layer consists of patterns like *shared hash table* or *distributed array.* On the low-level, *concurrent execution patterns* (formerly *parallel execution patterns*) are introduced. They are also split into two categories, that is *Advancing Program Counters* patterns like *dataflow* and *speculation*, and *coordination* patterns like *message passing* and *transactional memory.*

6.2. Parallel Design Patterns for Hard Real-Time Programs

So far, parallelisation approaches for HRT programs are still in an early research phase. One approach is currently investigated in the EU-project parMERASA[2], and in the following the preliminary baseline of that approach is presented. The parallelisation approach is split into two layers to meet the specific needs of real-time systems. On the lower level *synchronisation idioms*, that is platform-dependent solutions to ease the computation of WCET guarantees for synchronisations, are introduced, while on the higher level *parallel design patterns* provide platform-independent practices to form the overall structure of timing predictable parallel HRT programs. However, (parallel) design patterns provide reusable solution to recurring problems, hence, they allow for variability in the solution, meaning that not only one distinct solution might be derived from them, but different classes of solutions are possible. So, applying parallel design patterns might still allow for too much variation in the resulting parallel HRT program to enable the computation of tight WCET guarantees with static timing analysis tools.

Therefore, one issue, which is so far not accounted for in the presented preliminary parallelisation approach, is to further reduce the possible variability of solutions to tighten the possible design space to more distinct structures that can be more easily analysed with static timing analysis tools. But, design patterns are limited in providing such tight structures, without losing the reusability property. Thus, it might be advantageous to expand the pattern-based parallelisation approach by an additional layer between the higher-level of applying design patterns and the lower level of platform-dependent synchronisation technique.

A possible additional layer or additional part of parallel design patterns, currently under investigation in the parMERASA project, might be formed from algorithmic skeletons (see Jahr et al. 2013a,b, and also Sections 6.2.2 and 6.3 for more details).

[1]see also Our Pattern Language (OPL), a pattern language for parallel programming (version 2.0) at `http://parlab.eecs.berkeley.edu/wiki/patterns/patterns` [last accessed: April 2013]
[2]see parMERASA project website at `www.parmerasa.eu` [last accessed: April 2013]

6.2.1. Meta-Patterns

In the following, meta-patterns for the two layers of the real-time capable parallelisation approach are shown. They are based on the structure proposed by Mattson et al. (2004) and Keutzer et al. (2010), who describe the following scheme as meta-pattern (e.g. for their pattern language *OPL*[1]):

1. Name
2. Problem
3. Context
4. Forces
5. Solution
6. Invariants
7. Example
8. Known uses
9. Related patterns
10. References
11. Authors

A meta-pattern provides the skeleton for writing design patterns in a form that is reused for all patterns (of a pattern catalogue, system or language). This eases the usability, readability and application of those design patterns, while also allowing for experts to introduce new—and edit existing—design patterns.

Modifications to the above presented meta-pattern structure, for the approach depicted in the following, are introduced from a real-time perspective. That is how to include the mandatory real-time requirements for programmers, and how to include the possible output for (static) timing analysis tools, too. Also, the forces/motivation part includes motivational aspects that are necessary or helpful in the real-time domain. Furthermore, the meta-patterns for the introduced *parallel design patterns* and *synchronisation idioms* are slightly different, that is both meta-patterns are adjusted to their abstraction level. The meta-patterns should be well-defined to not enforce later modifications to their structure, as this would lead to rewriting all existing design patterns and idioms to fulfil those structure. However, changes in the design patterns and idioms itself are sincerely welcome, that is design patterns evolve and including best-practices from a wide range of experts promises to bring ahead the best solutions.

6.2.2. Meta-Pattern for Real-Time Parallel Design Patterns

The proposed meta-pattern scheme for parallel design patterns is based on state-of-the-art meta-pattern, as e.g. the one proposed by Keutzer et al. (2010), Mattson et al. (2004) (see 6.2.1). Additional items (6-8) are introduced to respect requirements arising from a real-time perspective and the impact on the static timing analysis. They provide information for programmers, and also highlight annotations and details that should or need to be provided by the programmer for the static timing analysis.

1. **Name**
   Give your design pattern a unique name which should reflect what the pattern does. Using unique names here helps to distinguish between patterns for discussions.
2. **Problem**
   State which parallelisation problem the design pattern solves.
3. **Context**
   Give examples and hints in which context this design pattern is helpful. For instance, in which real-time domain or for which specific real-time problems could this pattern be possible used (e.g. automotive domain, avionic domain, etc.)
4. **Forces/Motivation**
   Motivate why this design pattern is a good solution to the above problem. Also highlight if and how the design pattern helps to foster timing predictability.
5. **Solution**
   Present the solution, describe the design pattern in detail. If the solution leaves too much room for variance, check if it is better to split those into two separated design patterns to foster timing analysable program structures.
6. ***Real-Time Prerequisites***
   State which prerequisites are mandatory, and which requirements arise from real-time perspective.
7. ***Synchronisation Idioms***
   Give a list of synchronisation idioms which have to (can) be used for timing analysable data exchange or progress coordination when applying this pattern.
8. ***WCET Hints***
   Generate input for the WCET analysis. Give hints to the WCET analysis from what is decided by this design pattern, e.g. which parts of the parallelised code need to be annotated, and which information are needed in the annotation files.
9. **Example**
   Present an example (code and graphical representation) on how the design pattern is used on a fitting problem. Also show in the example what information is needed for the WCET analyses, e.g. how and which annotations are needed, or how the application might influence the timing behaviour.
10. **Known Uses**
    Put references to exemplary known uses of this design pattern.
11. **Related Patterns**
    Name related design patterns which might also be of interest for the programmer.
12. **References**
    Add references which are helpful for the programmer, including publications describing the algorithm in detail, prove correctness or analyse the timing behaviour.
13. **Authors**
    State your names and contact info.

The synchronisation idioms category (item 7) should provide a list of stand-alone idioms on synchronisation techniques. E.g. when using a specific parallel design pattern, the programmer might have different requirements on how the data between concurrently parallel HRT threads is exchanged, or how the progress is coordinated. That might be, for instance, a *last is best* strategy, or it might be required that the threads notify each other on each change of the data. Those different cases would then result in different synchronisation idioms. Also, the use of the synchronisation idioms, or in more detail their property of timing analysability, highly depends on the chosen ISA, the RTOS, and the programming model. For example it might be possible that a given platform supports the use of real-time transactional memory, whereas another platform requires lock-based synchronisation. So, each synchronisation idiom presents different timing analysable solutions, e.g. lock-based or non-blocking synchronisation techniques, on different platforms.

Also, as already discussed above, it could be advisable to further extend parallel design patterns with algorithmic skeletons to reduce the possible design space of resulting programs. One possibility would be to integrate them, similar to synchronisation idioms, as a separated item in each pattern. This could allow for providing platform-depended code skeletons that are applicable for the chosen design pattern.

Another possibility could be to integrate such programming skeleton in very specific examples (item 9), which then provide a tight structure, and only allow for adding user code in between given or generated code structures. Both approaches could be even further extended to build complete frameworks from—or on top of—specific design patterns, an approach which is for instance proposed by Asanovic et al. (2009) to support and ease the development of parallel programs for highly parallel platforms.

6.2.3. Meta-Pattern for Real-Time Synchronisation Idioms

The data exchange and progress coordination between threads of a parallel program at synchronisation points is an important factor concerning the estimation of WCET guarantees (e.g. worst-case waiting times). On the one hand, for being able to compute upper bounds at all, synchronisation techniques used at synchronisation points in a parallel program need to be designed for timing analysability, and mostly also need to be clearly understood by a static timing analyser or static timing analysis tool. On the other hand, the overestimation and pessimism introduced from synchronisations ought to be as low as possible to gain worst-case efficiency and performance, and timing predictable behaviour.

Synchronisation techniques are very specific for a given execution platform, ISA, and RTOS/system software, and the used programming model, that is mostly shared-memory or message passing. Therefore, the programmer should select the synchronisation techniques in dependence of those parameters. In the EU-project parMERASA[2], an extended pattern system (also including a number of different timing analysable synchronisation idioms for the parMERASA processor architecture) is in development. The proposed idioms should be used by application programmers as an interface for timing predictability of synchronisations.

Therefore, it is important to describe for an application programmer in detail (and with examples) what a specific synchronisation idiom does, whereas it is important for the static timing analysis to assure that the given synchronisation idiom is timing analysable. By using the given, specific synchronisation idioms, it is possible to reduce the pessimism, which arises in the static timing analysis when using non-standard synchronisations. And, even more severe, the use of non-standard, manually coded synchronisation constructs might lead to a situation in which it is not possible for a static timing analysis tool to compute a WCET guarantee at all, for instance when it is not possible to recognise the semantic of that manually coded synchronisation construct. An example for this is, when a programmer writes his own constructs for progress coordination, but is not aware that a solution exists, which already solves the same problem (e.g. a barrier), and for which it is known how to analyse its timing behaviour. Then, using the known, timing analysable barrier implementation would not change anything for the semantic of the parallel program, but fosters a static WCET analysis with as less pessimism as possible. Also, some synchronisation techniques might be preferred over others, depending on their timing behaviour or availability on a given platform respectively for a given RTOS, e.g. busy-waiting locks over blocking locks, or even transactions or non-blocking algorithms. Please note that due to the tight link of the synchronisation techniques to the chosen programming model, architecture, and even the specific RTOS/system-software, the categorisation as idioms fits better than calling them synchronisation patterns.

In the following, a meta-pattern scheme for the synchronisation idioms is described with slight changes to the above introduced meta-pattern for the parallel design patterns.

1. **Name**
   Give your synchronisation idiom a unique name which should reflect what the idiom does.
2. **Problem**
   State which synchronisation/communication problem the idiom solves, e.g. data exchange or progress coordination.
3. **Solution**
   Present the solution the idiom provides and describe the idiom in detail.
4. **Requirements, Real-Time Prerequisites and WCET Recommendations**
   Describe what the specific requirements of the idiom are, and give details on what the programmer should keep in mind when using this idiom, e.g. specific coding guidelines (WCET recommendations). Also state which prerequisites are mandatory, and which ones arise from real-time perspective (WCET requirements).
5. **Implementations**
   Give implementation details on the idiom and examples on how they are used, e.g. describe in a list which programming models, architectures, RTOS versions, etc. have to be used, respectively are guaranteed to be analysable for a given platform/ISA/RTOS/...

Implementation Example:

a) **Programming Model:** shared-memory [message passing, ...] Pthreads [MPI, OpenMP, ...]
b) **ISA:** TriCore v1.3 [PowerPC v2.06, ...]
c) **Processor:** MERASA multi-core (Version T2) [Freescale P4080 (NSE1MMB), ...]
d) **RTOS:** MERASA system software (Version 1.0), [Wind River VxWorks (Version 6.9), ...]
e) **Types:** type_t,
Initialisation: init_function(type_t);
Functions: acquire_function(type_t), release_function(type_t)
f) **Pseudo-Code:** *Some lock function*

```
1: acquire_lock //Enter critical section
2: //Remainder critical section
3: ...
4: release_lock //Leave critical section
```

6. **WCET Annotation**
Generate input for the WCET analysis. Give annotations for the static WCET analysis from what is decided by this idiom that is for instance the number of cooperating or competing threads, ids at synchronisation points to refer from the source code to the annotation file, etc. If annotations depend on the above chosen implementation, state it here. If annotations require specific formatting for a given timing analysis tool, then add an example.
7. **Example**
Present an example (code and graphical representation) on how this idiom is used on a fitting synchronisation (data exchange/coordinate) problem, and how it is annotated for the WCET analysis.
8. **Known Uses**
Put references to exemplary known uses of this idiom in as most detail as possible.
9. **Related Synchronisation Idioms** Name related synchronisation idioms which might also be of interest for the programmer.
10. **References**
Add references which are helpful for the programmer. Also add e.g. references to specific coding guidelines which are relevant when using this idiom (ISA/RTOS/...).
11. **Authors**
State your names and contact info.

6.2.4. Real-Time Parallel Design Patterns (Layer 1)

In this section an example of a real-time design pattern, the *Periodic Task Parallelism Pattern*, is presented. In the EU-project parMERASA further parallel design patterns are currently under investigation, and the here presented example has been derived from a case study with Bauer Maschinen AG for a large drilling machine control code (Gerdes et al. 2011), and is still under development.

1. **Name**
   Periodic Task Parallelism Pattern
2. **Problem**
   A number of tasks are executed periodically, either after each other in a random order, or in a specific order. In the sequential case, the tasks are scheduled either without a specific period, that is in some priority order, or, they are scheduled by a given period. This period might be the same for all tasks; however, it is also possible that different tasks have different periods, e.g. arising from their deadlines. Often the tasks are executed inside a `while(true)`-loop (control loop) on a sequential processor, and therefore interrupt the code that is executed in the loop. Typically, the response time of tasks is an important factor.
3. **Context**
   This problem occurs in the domain of machinery control systems, e.g. the control code of large drilling machines of Bauer Maschinen (see [1]). In a control loop, the design and flow of the program is directly derived from the tasks.
4. **Forces/Motivation**
   Decomposing the sequential program into tasks is quite intuitive for such control loop programs as they mostly already provide such a task structure. Therefore, load balancing and distribution of tasks over a number of cores comes more or less naturally. Moving tasks from a single-core processor environment to a multi-core processor facilitates the potential of executing more tasks without increasing the response time of those tasks too much.
5. **Solution**
   Decompose the sequential program or problem into tasks. If a sequential version already exists, there are most likely tasks that are scheduled in a given order. These sequentially scheduled tasks could then be the tasks for the parallel version. However, if further task dependencies exist, they need to be taken into account. Also, it might be beneficial to further decompose specific computational intensive tasks, e.g. by applying the *Data Parallel Pattern*. For communication and data exchange between tasks, respectively synchronisation of tasks, only the below stated methods (7. Synchronisation) should be used for a given architecture/ISA/RTOS. Now, the WCET of each task can be computed; in a first step this can be done without accounting for worst-case communication and waiting times (or by just assuming architecture depending constant worst-case latencies). The WCETs of each task could then be used to account for a first mapping of tasks to threads/-cores and a schedulability analysis. With those WCETs as a baseline, a further agglomeration and load-balancing of tasks to cores might be needed.

In a next step, the communication vs. computation ratio can be computed. If possible, tasks that communicate very often or exchange much data should be agglomerated into one core (localisation), or having low (worst-case) latencies.

6. **Real-Time Prerequisites**
   The tasks need to be scheduled and mapped statically. To achieve an improved worst-case performance, the agglomeration of tasks to threads, and the mapping of threads to cores should be load-balanced on WCETs. Function calls which may potentially suffer high latencies, e.g. by accessing slow I/O devices, should be annotated for the parallelisation and also for the static timing analysis.
7. **Synchronisation Idioms**
   a) **Ticket locks (or other locking techniques)** can be used to secure shared access to resources and data by enforcing mutual exclusion, especially to enforce atomicity of shared data access (the usability of blocking or busy-waiting locking techniques depends on the platform and the analysis tool/method).
   b) **Semaphores** can be used to coordinate accesses to resources, e.g. I/O devices. Binary semaphores can be used to enforce mutual exclusion.
   c) **Barriers** can be used to establish an order in which tasks should be executed.
   d) **Read/write locks** can be used for access to shared data/resources allowing multiple concurrent readers, but only one writer.
   e) **Blackboards** can be used to allow for concurrent access when the accuracy of the read values is of less importance.
8. **WCET Hints**
   To compute an estimated WCET of every thread, the static timing analysis tool needs to know which tasks are agglomerated into one thread/core, and also where the data is located. Depending on the chosen synchronisation idioms, further annotations and WCET hints must be provided as specified in the corresponding *synchronisation idiom*.
9. **Example**
   An example is the parallelisation of the large drilling machine control code of Bauer Maschinen presented in [1], which was done in the EU-project MERASA (see `www.merasa.org`). The sequential program was split into several tasks. The new tasks have been derived directly from the sequential version (see Figure 1). Each task on the right side of Figure 1, that is the main task, pulse-width modulation (PWM) tasks, I/O tasks, and controller area network (CAN) tasks, have been handled in a task array for a software scheduler in the sequential version. For the parallelisation, these tasks have been agglomerated into threads, and then each thread was mapped to one core of the quad-core MERASA processor; Figure 2 depicts the distribution of selected tasks. The bottom of Figure 2 shows a parallel program version with included barriers after each thread has executed one task. By this, memory contention is relaxed, and the program with synchronised releases resembles the sequential version. However, the second step of load balancing was not completed. In that second step, load balancing and further agglomeration of tasks to threads could be done to increase the worst-case performance.

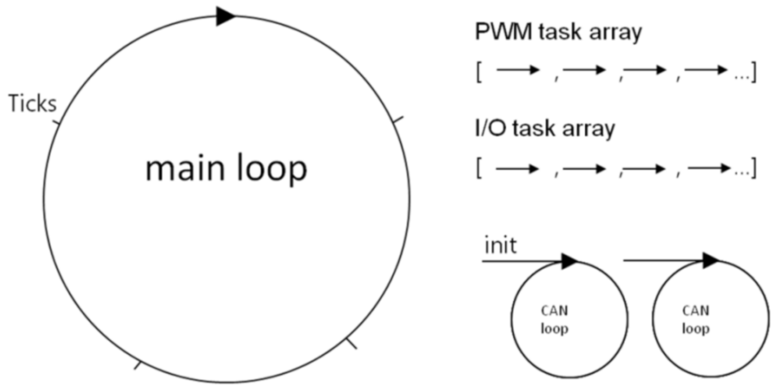


FIGURE 1: The sequential task structure of the large drilling machine control code (figure taken from [1]). On the left side: The main loop that is interrupted by a scheduler at specific times (Ticks). On the right side: The tasks and their classification in task arrays.

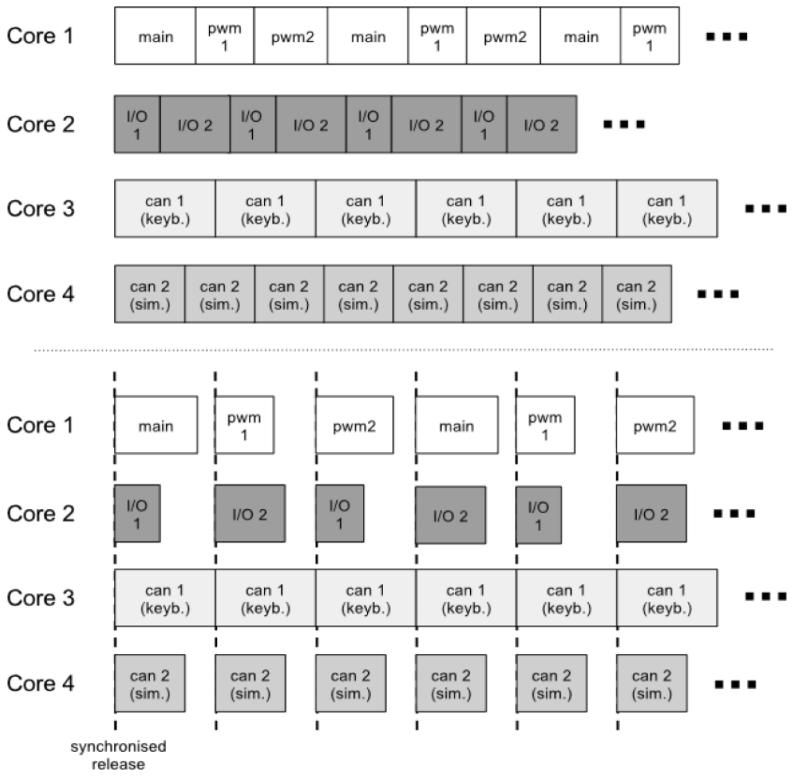


FIGURE 2: The distribution of tasks to the four cores of the MERASA quad-core processor (figure taken from [1]). Top: Unsynchronised execution of tasks. Bottom: Synchronised release of tasks enforced with barriers.

For instance, from Figure 2, the tasks *pwm1* and *I/O2* could be executed both on the same core, if the sum of the estimated WCETs of those two tasks would be smaller than the WCET of the longest task in that iteration, that is *can1*.
Remark: This example should be further explored and detailed (e.g. with the lessons learnt from case studies in the EU-project parMERASA).

10. Known Uses
Parallelisation of large drilling machine control code (see [1]).

11. Related Patterns

- *Task Parallelism Pattern* (high-performance domain) as stated in [2]
- *Embarrassingly Parallel Pattern* (high-performance domain) as stated in [3]
- *Data Parallelism Pattern*

12. References
[1] M. Gerdes, J. Wolf, I. Guliashvili, T. Ungerer, M. Houston, G. Bernat, S. Schnitzler, and H. Regler: Large Drilling Machine Control Code—Parallelisation and WCET Speedup. In 6th IEEE International Symposium on Industrial Embedded Systems (SIES), pages 91-94, June 2011.
[2] T. Mattson, B. Sanders, and B. Massingill: Patterns for parallel programming. Addison-Wesley Professional, 1st Edition, 2004. – ISBN 0321228111
[3] Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders: Parallel programming with a pattern language. International Journal on Software Tools for Technology Transfer (STTT), Volume 3, Number 2, pages 217-234, 2001.

13. Authors
Mike Gerdes (gerdes@informatik.uni-augsburg.de)
Ralf Jahr (jahr@informatik.uni-augsburg.de)
Andreas Hugl (Bauer Maschinen AG)

6.2.5. Real-Time Synchronisation Idioms (Layer 2)

As an example of synchronisation idioms, the ticket lock idiom is presented below. It includes the implementation used in this thesis for the shared-memory, multi-core MERASA processor, and the implementation for the parMERASA processor as well (item 5). The parMERASA implementation is still preliminary; it needs to be updated when the system software is finally released. The requirements and real-time prerequisites (item 4), as well as the implementation category (item 5) should also contain information and proofs that the given implementation and real-time prerequisites hold (e.g. by referencing publications or even ISA manuals, if necessary). The presented WCET annotations (item 6) are still preliminary and are depending on the used timing analysis tool. In this case, a possible annotation format to be used with the OTAWA timing analysis tool has been assumed.

In general, it should be assured that the programmer catches the semantic and usage of the specific synchronisation idiom. As well, the timing analysers or timing analysis tools need to understand the implication of the used idiom on the (binary) code, so that it can be analysed (correctly).

1. **Name**
   Ticket Lock
2. **Problem**
   Ticket locks can be used as a fair spin lock mechanism in real-time systems to secure critical section and provide mutual exclusion [1].
3. **Solution**
   The semantic of ticket locks [2], based on Lamport's bakery algorithm [3], is as follows. Each thread gets a unique ticket id when trying to access a critical region (line 2 in pseudo code of implementation example 1). Threads are allowed to enter the critical region when their ticket id matches the current value of now served (line 3). The threads are busy-waiting, until their ticket id my ticket matches the value of now served. After a thread leaves a critical section, it increments now served (line 9), and the thread with the appropriate ticket id can now enter the critical section. The atomic incrementing of ticket id and now served id can be done with the F&I primitive. Thus, ticket locks implement a busy-waiting spin lock, which is, contrary to e.g. test-and-set spin locks, fair independently of the arbitration strategy in the memory interconnect in a shared-memory multi-core processor (e.g. in the MERASA processor).
4. **Requirements, Real-Time Prerequisites and WCET Recommendations**
   The given platform must allow for atomic and consistent use of RMW operations, that is e.g. a F&I primitive as in the implementation examples (see also [1]). The critical section secured with a ticket lock should be as short as possible.
5. **Implementations**

   Implementation Example 1:

   a) **Programming Model:** shared-memory (global address space), *Pthreads*[4]
   b) **ISA:** MERASA, based on TriCore v1.3.1 [5]
   c) **Processor:** MERASA multi-core (Version T2)
   d) **RTOS:** MERASA RTOS (updated version of [6])
   e) **Types:** `typedef uint32\_t ticket\_t`,
      Initialisation: `ticket\_lock\_init(ticket\_t *lock);`
      Functions: `static uint8_t ticket\_lock\_acquire(ticket\_t *lock)`, `static uint8_t ticket\_lock\_release(ticket\_t *lock)`
   f) **Pseudo-Code:** *Ticket lock with F&I*

```
1: //Enter critical section
2: my ticket = F&I(ticket_id)
3: while my_ticket != now_served do
5: end while
6: //Remainder critical section
7: ...
8: //Leave critical section
9: F&I(now_served)
```

Implementation Example 2:

a) **Programming Model:** (distributed) shared-memory (global address space), *Message Passing*
b) **ISA:** PowerISA v2.03 [7]
c) **Processor:** parMERASA multi-core (preliminary) [8]
d) **RTOS:** parMERASA system software (preliminary) [9]
e) **Types:** `typedef uint32\_t ticketlock\_t`,
Initialisation: `static void spin\_init (ticketlock\_t *lock);`
Functions: `static uint8\_t spin\_lock (ticketlock\_t *lock),`
`static uint8\_t spin\_unlock (ticketlock\_t *lock)`
f) **Pseudo-Code:** *Ticket lock with F&I*

```
1: //Enter critical section
2: my ticket = F&I(ticket_id)
3: while my_ticket != now_served do
5: end while
6: //Remainder critical section
7: ...
8: //Leave critical section
9: F&I(now_served)
```

g) **Remark:** The parMERASA system software currently only allows for one type of spin lock, therefore the function call `spin_lock` invokes a ticket lock mechanism.

6. WCET Annotation

Annotate the entry code in every thread competing for a ticket lock with the same unique ID and maximum number of threads competing for that lock.

Example annotation for OTAWA [10]:

`// $OTAWA$ $UID$ 234, $num_threads$ 4`

7. Example

For implementation example 1 (MERASA platform):

```
// Initialisation (only done by one thread)
ticket_t spatial_lock;
ticket_lock_init(spatial_lock);
// Declaration of shared variables
uint32_t i_am_shared_counter = 0;
...
uint32_t my_counter = 0;
// parallel code section executed by 4 threads
ticket_lock_acquire(spatial_lock); // $OTAWA$ UID=234, num_threads=4
i_am_shared_counter += 4;
my_counter = i_am_shared_counter;
ticket_lock_release(spatial_lock); // $OTAWA$ UID=234, num_threads=4
...
```

8. **Known Uses**
   MERASA RTOS, parMERASA system software, Linux Kernel since version 2.6 (same semantic, but non-real-time implementation for x86 architectures)
9. **Related Synchronisation Idioms**
   glsfd Spin Locks, TAS Spin Locks, Mutex Locks, Binary Semaphores
10. **References**
[1] Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, Pascal Sainrat: Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In Proceedings of Design, Automation and Test in Europe (DATE'12), pages 671-676, 2012.
[2] John M. Mellor-Crummey, Michael L. Scott: Synchronization Without Contention. In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91), pages 269-278, 1991.
[3] Leslie Lamport: A New Solution of Dijkstra's Concurrent Programming Problem. In: Communications of the ACM, Volume 17, Number 8, pages 453–455, August, 1974.
[4] POSIX 2008: IEEE Std 1003.1, 2008 Edition. The Open Group Base Specifications Issue 7, 2008.
[5] Infineon Technologies AG: TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1. http://www.infineon.com/dgdl/tc_v131_instructionset_v138.pdf?folderId=db3a304412b407950112b409b6cd0351&fileId=db3a304412b407950112b409b6dd0352. January 2008.
[6] Julian Wolf, Florian Kluge and Irakli Guliashvili: Final System-Level Software for the MERASA Processor. Technical Report No. 2010-08, Institute of Computer Science, University of Augsburg, http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/1451, October 2010.
[7] Power.org: Power Instruction Set Architecture v2.03. http://www.power.org/resources/reading/. September 2006.
[8] see www.parmerasa.eu for deliverables and publications on the parMERASA hardware architecture.
[9] Christian Bradatsch and Florian Kluge: parMERASA Multi-core RTOS Kernel. Technical Report No. 2013-02, University of Augsburg, Department of Computer Science, http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/year/2013/docId/2230, February 2013.
[10] see www.otawa.fr for more details.
11. **Authors**
   Mike Gerdes (gerdes@informatik.uni-augsburg.de)

6.3. Related Work

In the following, parallelisation approaches and methodologies that are relevant to the embedded domain and real-time systems are presented (the Section 4.5 already presented related work with the scope on static timing analysis of parallel HRT programs in detail).

However, the first presented work in this section by Douglass (2006) does not really cover real-time capable parallelisation or multi-core processors as a target, but presents design patterns for real-time systems, which are also relevant for the parallel design pattern layer of the above presented preliminary approach. Though, Douglass (2006) also presents design patterns that target concurrent execution, e.g. for access to shared resources and critical sections, the main goal is on providing design patterns using Unified Modeling Language (UML) to create software for embedded real-time systems with single-core processors. Beside that the presented design patterns of Douglass (2006) are general enough to be easy understandable and (re)usable, they do not enforce a tight structure to foster static timing analyses needed for HRT systems. That is Douglass' approach is not focused on HRT systems, HRT capable parallelisation for multi-core processors, nor timing analysability in detail, but on general software development in embedded systems. It eases the burden on programming and documenting through the use of UML, and the proposed design patterns are useful to be kept in mind when programming software used in real-time embedded systems. The presented preliminary approach in this thesis goes a step further: on the one hand, embedded multi-core processors are the target platform, and on the other hand a parallelisation approach using design patterns, skeletons, and idioms is envisioned to enforce tight structures to facilitate static timing analysability.

Cordes et al. (2010) present an approach for automatic decomposition and parallelisation of sequential programs for embedded multi-processor system-on-chips (MPSoCs). They use hierarchical task graphs, introduced by Girkar and Polychronopoulos (1994), as intermediate representation of a sequential program and then apply ILP to parallelise on a coarse-grained, task parallel level including the constraints of the embedded domain, e.g. power consumption or limited memory space. They abstract communication between different hierarchy levels through communication nodes to enable parallelisation of each node independently. In a later publication by Cordes and Marwedel (2012), the authors expand their parallelisation approach by employing genetic algorithms for multi-objective parallelisation. Beside easing the parallelisation of sequential programs with constraints from the embedded domain on a coarse-grained level, they do yet not cover the timing analysability of their approach. Also, it is (until today) highly unlikely, that automatic parallelisation approaches can produce fine-grained parallelisation structures, even on the low-level, that also demonstrate timing predictability and thus low WCET guarantees on embedded multi-core processors.

The preliminary parallelisation approach presented in this chapter is extended and gone after in the EU-project parMERASA[2]. It is based on parallel design patterns and synchronisation idioms, and involves the programmer *and* the timing analyser. Jahr et al. (2013a,b) show the current state of the general pattern-supported parallelisation approach and the implication for the programmer, while Ozaktas et al. (2013) show how requirements and prerequisites could be included in that approach from a timing analysis perspective. In detail, the parMERASA parallelisation approach mostly relies on parallelisation of a programmer with (deep) domain knowledge, and well-defined structures are enforced by application of only specifically allowed parallel design patterns, algorithmic skeletons and synchronisation idioms. Hence, timing analysability is guaranteed by enforcing well-known timing analysable structures, and needed WCET annotations for the static timing analysis. In the pattern-supported parallelisation approach, the WCET annotations and real-time requirements are integrated in the parallelisation approach. The parMERASA approach, similar to the approach by Cordes et al. (2010) and *UPPAAL*[3] (more details on UPPAAL are presented in Section 4.5), is also based on abstract representation of parallel programs. That is activity diagrams from UML (using UML is e.g. also proposed by Douglass (2006) for software development in the real-time systems' domain) are augmented an additional node type with parallel design patterns, called Activity Pattern Diagram (APD) (cf. Jahr et al. 2013a). Hence, the above presented parallelisation approach is applied to a UML model and APD of an existing sequential program, or used to create a UML model and activity diagram of a problem to be parallelised. Also, in contrast to the work of Cordes et al. (2010) and Cordes and Marwedel (2012), the parallelisation approach is split into two steps, and the first step is (mostly) not constrained by the underlying platform, but targeting a maximum possible parallelisation degree. Then, in a second step, the parallelisation degree is adjusted to a reasonable level concerning the worst-case performance on a given platform while also the design space is tightened and structured by enforcing less variability through, e.g., algorithmic skeletons and platform-specific idioms.

Besides the availability of a wide range of (parallel) design patterns, the field of HRT capable synchronisation techniques for multi-core processors is still rather limited, and so far not well-defined. Reason for this are the high dependency on the underlying platform, the lack of parallel (multithreaded) HRT programs, and the fact that developing such programs is still in research (see also Section 3.3.7). However, in the domain of high-performance computing, such synchronisation techniques are already available in libraries and collections. As one example, the website `http://www.concurrencykit.org/` collects hardware and software synchronisation techniques, including non-blocking data structures, for a number of different platforms to ease the development and programming of high performance parallel programs. The same idea is pursued by providing synchronisation idioms in the parMERASA parallelisation approach, with the additional aim of providing only synchronisation techniques that are known to be timing analysable and predictable for a given (embedded) multi-core platform.

[3]see more details and publications on UPPAAL at `http://www.uppaal.org/` [last accessed: April 2013]

Time moves in one direction, memory in another. – William Gibson

7 Conclusion and Future Work

In this chapter the presented work of this thesis is summarised and concluded. Also, future work related to the key points of this thesis is shortly introduced.

7.1. Summary and Conclusion

The thesis on hand provides details on how to design timing analysable synchronisation techniques for parallel HRT programs on embedded shared-memory multi-core processors, and a preliminary approach towards pattern-supported HRT parallelisation. The proposed techniques have been implemented in the MERASA multi-core processor, and static timing analyses have been done with OTAWA, an open-source, static WCET tool.

In Chapter 3 the hardware-software co-design of HRT capable synchronisation techniques is presented in detail. The key aspect for those synchronisation techniques is to assure timing guarantees, that is upper bounds on the execution and waiting times for competing and cooperating HRT threads in a parallel program. The implemented software and hardware techniques need to fulfil specific requirements to achieve fairness on interfering accesses for being timing analysable with a static WCET tool on a shared-memory multi-core. Beside the *timing analysability* property, the techniques also aim to allow for tight upper bounds, that is *timing predictability* (see more details in Section 4.1.1). Therefore, the support for atomic memory access has been pursued by integrating the logic for RMW operations in an augmented memory controller. Thus, it is possible to enable atomicity and data consistency—under a *weak consistency* model(cf. Adve and Hill 1990)—without blocking the shared memory interconnect. Beside the gain in memory access time, this approach also fosters the portability of techniques to other (future) multi-core processors. For the handling of RMW operations, the implemented augmented memory controller reuses the `swap` instruction of the TriCore ISA. The implementation in the TriCore-based MERASA multi-core processor has been designed with minimum invasive changes allowing for also work with other processors' ISAs, as long as the reused instruction can be recognized in the memory controller. For instance, as detailed in Section 3.2.4, the `ldrex` instruction of the ARMv7-M ISA (2010) or the `lwarx` instruction of the PowerPC ISA (2010) are candidates for such modifications.

On top of the implemented RMW operations, namely TAS, F&I, and F&D, different lock-based software synchronisations have been implemented with regard to the requirements for timing analysable execution. On the one hand, busy-waiting spin locks and ticket locks are detailed, and, on the other hand, suspending locks (mutex locks and semaphores) are depicted (see Section 3.3). Additionally, different software barrier implementations are denoted in Section 3.3.6. Furthermore, a FIFO queue implementation using the F&I primitive has been introduced in Section 3.2.3. It is used to efficiently manage the waiting lists for threads in the blocking (binary) semaphore implementation, and to compare the impact of such a supporting structure versus the more traditional implementation of waiting lists with software-managed linked lists, which is used in the (fair) mutex lock implementation. The possibility to easily build FIFO queues with F&I stems from the cyclic-counting implementation of the F&I primitive: it increments a shared counter value until a given, priority initialised upper limit is reached, and automatically resets the counter value to '0'. A similar behaviour for decrementing has been implemented in the F&D primitive (see details in Section 3.2.2).

The *timing analysability* property of the proposed software synchronisations (relying on the different RMW operations) is then verified with a static WCET tool in Chapter 4 by formally computing upper bounds on the interferences of the different memory operations, namely normal load/store operations and RMW operations (see Section 4.2). Taking the FIFO order of memory operations in the augmented memory controller, and the round-robin arbitration between cores at the memory interconnect into account, it is possible to derive a WCML for each kind of memory operation. These WCMLs are then applied in the static WCET analyses of each software synchronisation method, detailed in Section 4.3. In the first analysis step, the synchronisation techniques are analysed independently of the program code to allow for some first impressions on their worse-case performance. The results in Section 4.3.4 show that busy-waiting locks exhibit a better worst-case performance than suspending locks. This stems from the analysis method, that is the WCET analysis cannot account for the lower interferences when threads are suspended. On the one hand, this is due to the employed SMT-core in the MERASA processor: if a HRT thread is suspended in one core, the other NHRT threads on that core might still be active and dispatch memory operations to the shared memory. On the other hand, the main issue is the limited knowledge of concurrent execution in a multi-core. When one thread is analysed, it is mostly not possible to make any assumptions about the execution state of other threads, hence, the worst-case scenario must be assumed to safely upper bound interferences.

The presented estimated WCETs also show that the (binary) semaphore implementation performs noticeably better than the (fair) mutex lock implementation, both, for the lock and unlock function. The reason is not the use of different spin lock techniques, e.g. TAS spin locks for the mutex locks and F&I/F&D spin locks for the (binary) semaphores, but the difference in the management of waiting threads. Mutex locks employ software-managed linked lists, and the semaphore implementation relies on the above mentioned FIFO queues with F&I (more details are discussed in Section 4.3.4). A similar positive effect of the F&I primitive can be observed from the comparison of F&I barriers versus subbarriers. While the subbarrier implementation uses conditional variables and mutex locks, the F&I barrier implements a light-weight logic which still ensures to be not prone to the reinitialisation problem (cf. Section 3.3.6). In comparison, the subbarrier implementation exhibits a more than five times higher WCET guarantee than F&I barriers. However, even more interesting is to compare the *timing predictability* or WCET tightness, that is the difference between the *real* WCET and the computed safe upper bound (the WCET guarantee). Regretfully, it is not possible to compute the real WCET; it is an NP-hard problem and the computation is prone to the halting problem. Therefore, to allow for making statements on *timing predictability* and WCET tightness, two benchmarking parallel programs have been used to evaluate the impact of different synchronisation techniques on their WCET guarantee.

Please note that this still does not allow for making precise statements on WCET tightness for the individual program execution. The problem therein is that by changing the programs, even by minor changes of just small code sections that include the acquire and release functions for locks, the real WCET might differ. Although that would be rather unlikely, the following example is conceivable: The WCET of a program version *A* with lock method *X* might be actually higher than for a program version *B* using lock method *Y*. Though, as this is not known, one cannot decide by the estimated WCETs of both individual program versions, which one exhibits tighter estimated upper bounds. In that case it might be possible that the WCET estimate of program version *A* is very high whereas it is very low for the other version. However, then it is still possible that the real but unknown WCET of program *A* is very close to its WCET estimated, but the estimated WCET of program version *B* differs vastly from its real *wcet*. Then, the assumption that a lower WCET estimate indicates a tighter WCET would be wrong. Nonetheless, following the definitions of Kirner and Puschner (2010) on timing predictability, one can then say that program version *A* exhibits better stability.

Taking these consideration into account, the two benchmarking programs, a parallel matrix multiplication and a parallelised *Integer Fast-Fourier-Transformation* (IFFT), are only slightly changed by either substituting the used locking method (for matmul and IFFT), or, for the case of IFFT, also the used barrier implementation. Then, it seems valid to assume that a lower WCET guarantee corresponds with an equal gain in WCET tightness, thus better *timing predictability*. Therefore, by comparing the improvement of WCET guarantees, choosing the program version with the highest estimated WCET as baseline, it should be feasible to conclude on which synchronisation technique achieves the lowest WCET guarantee, or, in other words, the highest improvement of WCET guarantees. Accordingly, from the results presented in Section 4.4, it can be concluded that busy-waiting ticket locks outperform the other suspending lock techniques. Ticket locks achieve an improvement of WCET guarantees of up to 1.52 compared to the same program version with (fair) mutex lock synchronisations. Additionally, in combination with F&I barriers, ticket locks achieve an improvement of up to 2.39 for the IFFT program in comparison to the version using mutex locks and subbarriers.

Motivated by the outcome of the computation of WCMLs—frequent normal load and store operations suffer from concurrent infrequent slower RMW operations—an optimisation technique to reduce the pessimism in the WCET analysis is introduced in Chapter 5: the split-phase synchronisation technique. To reduce the impact of slower (and infrequent) RMW operations in the WCMLs of frequent memory operations, and thus the resulting pessimism in the WCET estimates, the RMW operations are split into three phases in the augmented memory controller: a load phase, a modification phase, and a store phase.

The idea of splitting RMW operations allows for executing more frequent normal load/store operations in between the load and store phase of a RMW operation in the augmented memory controller. Thus, as shown in Section 5.3.1, the WCMLs of load and store operations are reduced, which is beneficial for the WCET guarantees of parallel HRT programs. Even so, it must be assured that splitting atomic memory operations does not introduce any data inconsistencies or destroy the atomic property.

Therefore, it is shown in Sections 5.2 and 5.2.4 that atomicity and consistency for the split-phase synchronisation technique can be retained under the *weak consistency* model with specific (hardware) solutions in the augmented memory controller. The split-phase technique has been exemplary implemented in the MERASA processor, but, just as well as for the synchronisation logic of the augmented memory controller, the split-phase technique should be applicable for further multi-core processors as well.

However, infrequent RMW operation now suffer from higher WCMLs. To still achieve better WCET guarantees for parallel programs with the split-phase technique, the impact of higher WCMLs for RMW operation must be taken into account (see Section 5.3.2). It can be shown that the split-phase technique is beneficial, if, depending on the number of cores and memory latency, less than a specific minimum percentage of all executed memory operations in the worst-case path of a program are RMW operations (on synchronisation variables). Beside counting normal memory operations versus RMW operations in the source code of a parallel program gives first hints on the possible usability of the split-phase technique, these occurrences do not reflect precisely in the executed worst-case path. Therefore, firstly the OTAWA tool has been enhanced (by the colleagues at University of Toulouse who maintain the OTAWA tool) to output the precise occurrences of RMW operations and total memory operations in the worst-case path of an analysed parallel program. And, secondly the two benchmarking parallel programs have been analysed again with the split-phase technique and the corresponding WCMLs.

Results of WCET guarantees and improvements with the split-phase technique, and the ratio of RMW operations in their worst-case path are presented in Section 5.3.3. In detail, the ticket lock configurations achieve the highest gain—up to an improvement of WCET guarantees of 1.3—with the split-phase technique. Surprisingly, the lowest gain is attained for the (binary) semaphores (1.03 to 1.15), being slightly lower than for the program version with mutex locks (1.08 to 1.18). This is then explained by taking the ratios of RMW operations over total memory operations into account: for ticket locks, the ratio is the lowest (2.82 % to 12.67 %), whereas for semaphores it is the highest (16.80 % to 32.16 %). Combining the results achieved with the worst-case efficiency of ticket locks and F&I barriers in concert with the split-phase synchronisation technique, a total WCET guarantee improvement of 2.8 for IFFT, respectively 2.0 for matmul, can be reached.

In summary, all program versions with each lock-based synchronisation and each barrier implementation achieve better WCET guarantees with the split-phase technique, compared to the WCET guarantees without the split-phase technique (as derived in the analysis in Section 4.4). That is the needed minimum percentage of non-RMW operations is exceeded for all configurations of the two parallel benchmark programs. Keeping in mind that the two programs do exhibit a high ratio of synchronisations, it is assumed that similar or even better results are achieved for other, real-world multithreaded parallel HRT programs with the split-phase synchronisation technique.

In Chapter 6 an introduction to a novel parallelisation approach for HRT programs targeting multi- and many-core platforms is presented. The parallelisation approach is envisioned as a two-layered process, based on parallel design patterns on a higher, platform-independent abstraction layer, and employing platform-dependent, timing analysable synchronisation idioms on the lower layer. Inspired from parallelisation approaches in the high-performance domain (cf. Foster 1995, Mattson et al. 2004), aims to achieve maximum possible parallelisation in a first step, which is then refined and tailored to a reasonable level on specific target platforms in a second step. But, instead of average-case performance in traditional parallelisation approaches, the approach targets timing predictability and worst-case performance. Therefore, the pattern-supported approach embraces the domain and program knowledge of software engineers and the requirements and guidelines provided from the perspective of static timing analysis to create timing analysable parallel HRT programs. Thus, information helpful for either the programmer or the timing analyser are collected and transferred between the two domains through real-time prerequisites and WCET hints in the design patterns and idioms on the one hand, and WCET annotations on the other hand. Additionally to the presented design patterns and idioms depicted in Sections 6.2.4 and 6.2.5, further patterns and idioms are collected and examined.

A possible weakness in the current state of the parallelisation approach could be that it still allows for a too wide range of potential resulting parallel program structures. A conceivable solution might be introducing further elements in the process, e.g. algorithmic skeletons, to further tighten the resulting variety of program structures (see also further discussions in the Section 7.2 on future work below).

The pattern-supported parallelisation approach is currently under investigation and further refinement in the EU-project parMERASA. Together with industrial partners from the automotive, avionic, and construction machinery domain the target is to achieve a well-defined and sound software engineering process for the parallelisation of HRT programs for future predictable many-core platforms.

7.2. Future Work

One interesting research direction, already discussed shortly in Section 3.3.7, is the use of optimistic concurrency control for parallel HRT programs in embedded multi- and many-core processors. Optimistic concurrency control, that is non-blocking techniques to access (large) data structures or transactional memory, promises, on the one hand, to ease the burden on programming and designing parallel programs. While, on the other hand, such techniques could also be used to allow for better integration of failure handling in the execution of parallel programs on multi-core architectures. Beside failure handling is out of scope in this thesis, it is a mandatory property to be satisfied in industrial HRT systems. Transactional memory and non-blocking techniques both promise such behaviour, even possibly allowing for very fine-grained failure handling and less expensive redundancy than today available redundant systems provide (e.g. lockstep execution (cf. Mukherjee 2008, p. 212f.) provided by the recently released Infineon AURIX (2013) multi-core processor for the automotive domain).

Techniques and software engineering approaches that help in the complex task of writing functional correct parallel programs are warmly welcome, especially when the additional requirements of timing correctness of embedded (hard) real-time systems are concerned. Chapter 6 introduces first steps for such an approach, however, until now it is mostly based on lock-based techniques to secure (short) critical sections by applying timing analysable synchronisation idioms. With an increasing amount of possible future programs to be parallelised, further techniques for worst-case efficient access to data structures might be satisfied using non-blocking (wait-free, lock-free, or obstruction-free) techniques, or even transactional memory. Research on using such techniques for embedded real-time systems already started (see Section 3.3.7 for a short survey on current research topics), however, yet it is still unknown if they also are timing analysable and worst-case efficient for multithreaded parallel HRT programs on shared-memory multi-core processors using static timing analysis tools. It might be interesting to quantise if and in which cases non-blocking techniques show better worst-case efficiency over lock-based techniques in shared-memory multi-core processors, e.g. depending on the size of critical sections or shared data structures.

Also, future research on further easing the manual effort for static timing analysis of parallel programs is a key point to establish such techniques, not only in academia, but also for industrial use. One very challenging but interesting research field sparks from the uncertainty on simultaneous execution of threads in a multi-core processor. Finding techniques to incorporate information on those unknown execution states could ease the problem of overestimation and pessimism in static timing analysis, e.g. techniques similar to the concepts of *timing barriers* (see Kirner and Puschner 2010) or *temporal firewalls* (see Kopetz and Nossal 1997).

One other future work concerns the proposed pattern-based parallelisation approach presented in Chapter 6. In the EU-project parMERASA this approach is researched in more detail in the following years, and focuses on providing a programming framework to engineer timing analysable parallel HRT programs, e.g. for exemplary industrial programs from the automotive, avionic, and construction machinery domain. Research on integrating the pattern-supported parallelisation approach on the higher abstraction levels in the well-known modelling language UML has already started (see Jahr et al. 2013a,b), and possible lock-based synchronisation techniques on the lower level have been proposed in this thesis. But, future research is needed to further tighten the possible design space and variability of resulting parallel programs to allow for timing predictable worst-case execution on many-core platforms. That is, as already denoted in Section 6.2, the link between high level design patterns on abstract representation, i.e. models, of a problem to be parallelised, and low-level synchronisation support needs to be filled. Yet, it seems promising to integrate algorithmic skeletons in the parallelisation approach, to enforce timing predictable parallel program structures. One other solution to that problem might be inspired from the proposals of Asanovic et al. (2009) for high-performance parallelisation. Asanovic et al. (2009) envision future parallel programming frameworks that are e.g. derived from just one parallel design pattern. As example they mention the very successful *Ruby on Rails* framework for single-core processors that is based on the *Model-View-Controller* pattern.

A Appendix: Source Code

A.1. Software Synchronisations in the MERASA RTOS

This Section lists the source code of implemented timing analysable software synchronisations as part of the MERASA RTOS. Each listing presents the `lock()` and `unlock()` functions for locking synchronisations, as well as functionalities used for barriers. Also, declarations of global variables or helper functions are added where necessary.

If not mentioned differently (e.g. for the subbarier implementation in Listing A.6) all the below published source code is made available under the *New BSD License*:

Copyright (c) 2008-2013 Mike Gerdes, Julian Wolf, Florian Kluge
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Augsburg nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Listing A.1:
Source Code: Spin lock implementation with test-and-set in the MERASA RTOS

```
static inline uint32_t cc_sl_init(cc_spinlock_t *lock) {
  *lock = 0;
  return 0;
}

static inline uint32_t cc_sl_trylock(cc_spinlock_t *lock) {
  uint32_t r = 1;
  #ifndef GCC
  __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock) :
"memory");
  #else
  asm volatile ("swap.w␣[%2]0,␣%0" : "=d" (r) : "0" (r), "a" (lock) :
"memory");
  #endif

  return r;
}

static inline uint32_t cc_sl_lock(cc_spinlock_t *lock) {
  while (cc_sl_trylock(lock));
  { }
  return 0;
}

static inline uint32_t cc_sl_unlock(cc_spinlock_t *lock) {
  uint32_t r = 0;
  #ifndef GCC
  __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock) :
"memory");
  #else
  asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r) :
"memory");
  #endif
  return 0;
}
```

Listing A.2:
Source Code: Spin lock implementation with fetch-and-increment/fetch-and-decrement in the MERASA RTOS

```
static inline uint32_t fd_lock(uint32_t *lock) {
  while(!fetch_and_decrement(lock))
  { }
  return 0;
}

static inline uint32_t fd_unlock(uint32_t *lock) {
  fetch_and_increment(lock);
  return 0;
}


static inline uint32_t fetch_and_add_init(uint32_t *data, uint32_t limit) {
  *data = limit<<16;
  return 0;
}


static inline uint32_t fetch_and_increment(uint32_t *address) {
  uint32_t data = 0x0000FFFF;  // id for the augmented memory controller
  #ifndef GCC
  __asm("swap.w␣%0,[%2]0" : "=d" (data) : "0" (data), "a"
((uint32_t*) address) : "memory");
  #else
  asm volatile ("swap.w␣[%2]0,␣%0" : "=d" (data): "0" (data), "a"
(address) : "memory");
#endif

  return data & 0x0000FFFF;
}

static inline uint32_t fetch_and_decrement(uint32_t *address) {
  uint32_t data = 0x70007000; // id for the augmented memory controller
  #ifndef GCC
  __asm("swap.w␣%0,[%2]0" : "=d" (data) : "0" (data), "a"
((uint32_t*) address) : "memory");
  #else
  asm volatile ("swap.w␣[%2]0,␣%0" : "=d" (data): "0" (data), "a"
(address) : "memory");
  #endif

  return data & 0x0000FFFF;
}
```

Listing A.3:
Source Code: Ticket lock implementation in the MERASA RTOS

```
typedef struct {
  volatile uint32_t next_ticket;
  volatile uint32_t now_serving;
} ticket_t;

static uint32_t ticket_lock_init(ticket_t *ticket) {
  ticket->next_ticket = (ticket_t)(0x7FFF0000);
  ticket->now_serving = (ticket_t)(0x7FFF0000);
  return 0;
}

static inline uint8_t ticket_lock_acquire(volatile ticket_t *ticket) {
  ticket_t *my_ticket;
  my_ticket->next_ticket = fetch_and_increment(ticket->next_ticket);
  my_ticket->serving;

  do {
    my_ticket->serving = (ticket_t)((*ticket->now_serving) & 0x0000FFFF);
  }
  while(ticket->my_ticket != my_ticket->serving);
  { }

  return 0;
}

static inline uint8_t ticket_lock_release(volatile ticket_t *ticket) {
  fetch_and_increment(ticket->now_serving);
  return 0;
}
```

Listing A.4:
Source Code: (Fair) Mutex lock implementation with TAS in the MERASA RTOS

```
typedef struct thread_control_block_t tcb_t;
typedef thread_handler pthread_t;
typedef uint32_t sched_t;

typedef struct pthread_mutex {
        volatile uint32_t the_lock;
        cc_spinlock_t guard;
        thread_handler owner;
        sched_t prev_sched;
        /*!< Scheduling parameters before the guarded critical block */
        tcb_t *waitlist_first_out;
        tcb_t *waitlist_last_out;
} pthread_mutex_t;

int32_t ccthread_mutex_init(pthread_mutex_t* mutex) {
  mutex->the_lock = 0;
  cc_sl_init(&mutex->guard);
  mutex->owner = NO_OWNER;
  mutex->waitlist_first_out = NULL;
  mutex->waitlist_last_out = NULL;
  return 0;
}

int32_t ccthread_mutex_lock(pthread_mutex_t* mutex) {
  thread_handler th = get_current_thread_handler();
  cc_sl_lock(&mutex->guard);
  threadptr thread = get_thread4handler(th);
  tcb_t *my_tcb = TH2TCB(th);

  while (mutex->the_lock == 1) {
  // Thread into waiting list, and placed behind other waiting threads
  if (mutex->waitlist_last_out != NULL) { // not the first in list

    // connect with other threads
    mutex->waitlist_last_out->synclist_next = my_tcb;
    my_tcb->synclist_prev = mutex->waitlist_last_out;

    // set this thread as last thread in the whole list
    mutex->waitlist_last_out = my_tcb;
    // waitlist_first_out is already set
    }
    else { // thread is the first one in list
    mutex->waitlist_first_out = my_tcb;
    mutex->waitlist_last_out = my_tcb;
    }
    // now go to sleep
    _tie();
    set_suspended(my_tcb);
    cc_sl_unlock(&mutex->guard);
    _untie();
    // after wakeup gain local spinlock (guard)
    cc_sl_lock(&mutex->guard);
    #ifdef FAIR_MUTEX
    /* In the fair mutex implementation, the thread unlocking the mutex
     * does not set mutex->the_lock = 0, as a thread that is already
     * waiting for the mutex should get it, not a thread that might be
     * trying to get it (Enforcing FIFO). */
    mutex->owner = th;
    cc_sl_unlock(&mutex->guard);
    return 0;
    #endif
  }
  // lock gained
  mutex->the_lock = 1;
  mutex->owner = th;
  cc_sl_unlock(&mutex->guard);
  return 0;
}


int32_t ccthread_mutex_unlock(pthread_mutex_t* mutex) {
  thread_handler th = get_current_thread_handler();
  if ((mutex->owner != th) && (th != SYSTEM_THREAD_HANDLER)) {
    return EPERM;
  }
  cc_sl_lock(&mutex->guard);
  if (mutex->waitlist_first_out != NULL) {  // there are threads waiting

    tcb_t *tcb_to_unsuspend = mutex->waitlist_first_out;
    if (tcb_to_unsuspend->synclist_next != NULL) {
      // there is more than one thread waiting

      // set second thread in waitlist as "first out"
      mutex->waitlist_first_out = tcb_to_unsuspend->synclist_next;
      // remove connections to second thread
      mutex->waitlist_first_out->synclist_prev = NULL;
    }
    else { // there is only one thread waiting, so set list to NULL
      mutex->waitlist_first_out = NULL;
      mutex->waitlist_last_out = NULL;
    }
    tcb_to_unsuspend->synclist_next = NULL;
    unset_suspended(tcb_to_unsuspend);

    #ifdef FAIR_MUTEX
    /* In the fair mutex implementation, the thread unlocking the mutex
     * does not set mutex->the_lock = 0, as a thread that is already
     * waiting for the mutex should get it, not a thread that might be
     * trying to get it (Enforcing FIFO). */
    cc_sl_unlock(&mutex->guard);
    return 0;
    #endif
  }
  mutex->owner = NO_OWNER;
  mutex->the_lock = 0;
  cc_sl_unlock(&mutex->guard);

  return 0;
}

int32_t ccthread_mutex_trylock(pthread_mutex_t* mutex) {
  cc_sl_lock(&mutex->guard);
  int32_t rv = E_OK;
  if(mutex->the_lock == 0) {
    thread_handler th = get_current_thread_handler();
    mutex->owner = th;
  }
  else {
    rv = EBUSY;
  }
  cc_sl_unlock(&mutex->guard);
  return rv;
}
```

Listing A.5:
Source Code: Semaphore implementation with
fetch-and-increment/fetch-and-decrement in the MERASA RTOS

```
typedef struct {
        uint32_t value;
        uint32_t temp;
        uint32_t waitlist_entries;
        uint32_t waitlist_lock;
        uint32_t fifo_next;
        uint32_t fifo_last;
        tcb_t *waitlist_fifo[NO_CORES];
} pthread_sem_s_t;

static inline uint32_t pthread_sem_init_suspend(
  pthread_sem_s_t *sem,
  int32_t pshared,
  uint32_t value) {
        sem->value = value<<16 ^ value;
        fetch_and_add_init(&sem->waitlist_lock,1);
        fetch_and_increment(&sem->waitlist_lock);
        fetch_and_add_init(&sem->fifo_next,pshared);
        fetch_and_add_init(&sem->fifo_last,pshared);
        fetch_and_add_init(&sem->waitlist_entries,pshared);

        uint32_t i;
        for(i=0; i < pshared; i++) {
                sem->waitlist_fifo[i] = NULL;
        }
        return 0;
}

static inline int32_t pthread_sem_wait_suspend(pthread_sem_s_t *sem) {
        if((fetch_and_decrement(&sem->value)) <= 0) {
                /* sleep and enter waiting list;
                waking up from other thread calling sem_post*/
                while(!fetch_and_decrement(&sem->waitlist_lock))
                { }
                if((fetch_and_decrement(&sem->value)) <= 0) {
                        thread_handler th = get_current_thread_handler();
                        tcb_t *my_tcb = TH2TCB(th);
                        sem->waitlist_fifo[
                        fetch_and_increment(&sem->fifo_next)] = my_tcb;
                        fetch_and_increment(&sem->waitlist_entries);

                        _tie();
                        set_suspended(my_tcb);
                        fetch_and_increment(&sem->waitlist_lock);
                        _untie();

                        fetch_and_increment(&sem->waitlist_lock);
                        return 0;
                }
                else {
                        fetch_and_increment(&sem->waitlist_lock);
                        return 0;
                }
        }
        return 0;
}

static inline int32_t pthread_sem_post_suspend(pthread_sem_s_t *sem) {
        while(!fetch_and_decrement(&sem->waitlist_lock))
        { }

        if(fetch_and_decrement(&sem->waitlist_entries)) {
                tcb_t *tcb_to_unsuspend = (tcb_t*)sem->waitlist_fifo[
                fetch_and_increment(&sem->fifo_last)];
                unset_suspended(tcb_to_unsuspend);

                return 0;
        }
        else {
                fetch_and_increment(&sem->value);
                fetch_and_increment(&sem->waitlist_lock);
                return 0;
        }
        return 0;
}
```

Listing A.6:
Source Code: Subbarrier implementation in the MERASA RTOS (based on the source code from the LEON OpenSPARC simulator (http://kenai.com/projects/legion/sources/legion-opensparc/content/src/generic/barrier.c?rev=7) which in turn is based on a version from Marejka (1994))

```
/*
* OpenSPARC T2 Processor File: barrier.c
* Copyright (c) 2006 Sun Microsystems, Inc.  All Rights Reserved.
* DO NOT ALTER OR REMOVE COPYRIGHT NOTICES.
*
* The above named program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public
* License version 2 as published by the Free Software Foundation.
*
* The above named program is distributed in the hope that it will be
* useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
* General Public License for more details.
*
* You should have received a copy of the GNU General Public
* License along with this work; if not, write to the Free Software
* Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.
*/

int barrier_init(pthread_barrier_t *barrier, int needed) {
        int32_t n = 0;
        uint32_t i = 0;

        if (needed < 1)
          return EINVAL;

        barrier->maxcnt = needed;
        barrier->subbarrierp = &barrier->subbarrier[0];

        for (i = 0; i < 2; i++) {
                struct _subbarrier *subbarrierp = &(barrier->subbarrier[i]);
                subbarrierp->runners = needed;

                if (n = pthread_mutex_init(&subbarrierp->wait_lock, NULL)) {
                        return n;
                }

                if (n = pthread_cond_init(&subbarrierp->wait_cond, NULL)) {
                        return n;
                }
        }
        return 0;
}
```

```
int barrier_wait(pthread_barrier_t *barrier) {
        struct _subbarrier *subbarrierp = barrier->subbarrierp;

        pthread_mutex_lock(&subbarrierp->wait_lock);

        if (subbarrierp->runners == 1)  {
                // last thread to reach barrier
                if (barrier->maxcnt > 1) {
                        // reset runner count and switch subbarriers
                        subbarrierp->runners = barrier->maxcnt;
                        barrier->subbarrierp = (barrier->subbarrierp ==
                                                &barrier->subbarrier[0])
                        ? &barrier->subbarrier[1]  : &barrier->subbarrier[0];

                        // wake up waiting threads
                        pthread_cond_broadcast(&subbarrierp->wait_cond);
                }
        }
        else {
                subbarrierp->runners--; // one less running thread

                while (subbarrierp->runners != barrier->maxcnt) {
                        pthread_cond_wait(&subbarrierp->wait_cond,
                        &subbarrierp->wait_lock);
                        // wait until last thread to reach barrier
                }
        }

        pthread_mutex_unlock(&subbarrierp->wait_lock);

        return 0;
}
```

Listing A.7:
Source Code: Barrier implementation with fetch-and-increment in the MERASA RTOS

```
typedef struct {
  uint32_t needed;
  uint32_t runners;
  uint32_t waitlist_lock;
  tcb_t *waitlist_fifo[NO_CORES];
} pthread_barrier_t;

int barrier_init(pthread_barrier_t *barrier, int needed) {
  if(needed < 1) {
    return EINVAL;
  }
  fetch_and_add_init(&barrier->runners,needed -1);
  barrier->needed = needed;

  fetch_and_add_init(&barrier->waitlist_lock,1);
  fetch_and_increment(&barrier->waitlist_lock);

  uint32_t i;
  for (i=0; i < needed-1; i++) {
    barrier->waitlist_fifo[i] = NULL;
  }
}

int barrier_wait(pthread_barrier_t *barrier) {
  while(!fetch_and_decrement(&barrier->waitlist_lock))
  { }
  uint32_t cur_runner = fetch_and_increment(&barrier->runners);
  uint32_t needed = (uint32_t) barrier->needed;

  if( cur_runner >= (needed - 1)) {
  // last needed thread to reach barrier
    uint32_t i = 0;
    for(i = 0; i < needed-1; i++) {
        tcb_t *tcb_to_unsuspend = (tcb_t*)barrier->waitlist_fifo[i];
        unset_suspended(tcb_to_unsuspend);
      }
    fetch_and_increment(&barrier->waitlist_lock);
  }
  else {
    thread_handler th = get_current_thread_handler();
    tcb_t *my_tcb = TH2TCB(th);
    barrier->waitlist_fifo[cur_runner] = my_tcb;

    _tie();
    set_suspended(my_tcb);
    fetch_and_increment(&barrier->waitlist_lock);
    _untie();
  }
  return 0;
}
```

B Appendix: Binary Code

B.1. Binary Code of Software Synchronisations

The following binary codes are the compiled version of the source code in Appendix A.1 (Some output of the compiler has been omitted for visual reasons). The binary codes presented in this Appendix are the baseline for the static timing analyses of software synchronisations in Chapter 4.

If not mentioned differently all the below published binary code is made available under the *New BSD License*:

Copyright (c) 2008-2013 Mike Gerdes, Julian Wolf, Florian Kluge
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Augsburg nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Listing B.1:
Binary code: Spin lock with test-and-set

```
<cc_sl_lock>:
a0002c60:        82 1f            mov %d15,1
         // r = 1;
         #ifndef GCC
         __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
           : "memory");
         #else
         asm volatile ("swap.w␣[%2]0,␣%0" : "=d" (r) : "0" (r), "a" (lock)
           : "memory");
         #endif
a0002c62:        49 ff 00 08      swap.w [%a15]0,%d15

<cc_sl_unlock>:
a0002c92:        82 0f            mov %d15,0
         // r = 0;
         #ifndef GCC
         __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
           : "memory");
         #else
         asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r)
           : "memory");
a0002c94:        49 ff 00 08      swap.w [%a15]0,%d15
```

Listing B.2:
Binary code: Spin lock with fetch-and-increment/fetch-and-decrement

```
a0002a5e:        1b 00 00 f7      addi %d15,%d0,28672
         #ifndef GCC
         __asm("swap.w␣%0,[%2]0" : "=d" (data) : "0" (data),
           "a" ((uint32_t*) address) : "memory");
         #else
         asm volatile ("swap.w␣[%2]0,␣%0" : "=d" (data): "0" (data),
           "a" (address) : "memory");
a0002a62:        49 cf 00 08      swap.w [%a12]0,%d15
a0002a66:        bb f0 ff 1f      mov.u %d1,65535
a0002a6a:        26 1f            and %d15,%d1
a0002a6c:        6e f9            jz %d15,a0002a5e
```

Listing B.3:
Binary code: Ticket lock

```
<ticket_lock_acquire>
a00048a4:        91 b0 00 3a     movh.a %a3,40971
a00048a8:        bb f0 ff ff     mov.u %d15,65535
a00048ac:        d9 33 a4 b1     lea %a3,[%a3]6884 <a00b1ae4 <next_ticket>>
a00048b0:        49 3f 00 08     swap.w [%a3]0,%d15
a00048b4:        bb f0 ff 0f     mov.u %d0,65535
a00048b8:        26 0f           and %d15,%d0
a00048ba:        78 0b           st.w [%sp]44,%d15

a00048bc:        91 b0 00 2a     movh.a %a2,40971
a00048c0:        d9 22 b8 21     lea %a2,[%a2]6328 <a00b18b8 <now_serving>>
a00048c4:        4c 20           ld.w %d15,[%a2]0
a00048c6:        bb f0 ff 0f     mov.u %d0,65535
a00048ca:        26 0f           and %d15,%d0
a00048cc:        78 0a           st.w [%sp]40,%d15
a00048ce:        19 a0 2c 00     ld.w %d0,[%sp]44
a00048d2:        58 0a           ld.w %d15,[%sp]40
a00048d4:        5f f0 f4 ff     jne %d0,%d15,a00048bc <fix_fft_thread+0x44>


<ticket_lock_release>
a0004974:        bb f0 ff ff     mov.u %d15,65535
a0004978:        49 5f 00 08     swap.w [%a2]0,%d15
```

Listing B.4:
Binary code: (fair) Mutex lock with test-and-set

```
<pthread_mutex_lock>:
a0002c50 <ccthread_mutex_lock>:
a0002c50:       cd 92 00 0e     mtcr $0xe009 (unknown SFR),%d2
int32_t ccthread_mutex_lock(Pthread_mutex_t* mutex) {
a0002c54:       40 4c           mov.aa %a12,%a4
  thread_handler th = get_current_thread_handler();
a0002c56:       6d 00 5d 02     call a0003110 <get_current_thread_handler>
a0002c5a:       02 28           mov %d8,%d2
static inline uint32_t cc_sl_lock(cc_spinlock_t *lock) {
a0002c5c:       d9 cf 04 00     lea %a15,[%a12]4
a0002c60:       82 1f           mov %d15,1
a0002c62:       49 ff 00 08     swap.w [%a15]0,%d15     // lock(guard)
a0002c66:       ee fd           jnz %d15,a0002c60 <ccthread_mutex_lock+0x10>
a0002c68:       3c 25           j a0002cb2 <ccthread_mutex_lock+0x62>
  cc_sl_lock(&mutex->guard);
  threadptr thread = get_thread4handler(th);
  tcb_t *my_tcb = TH2TCB(th);
  while (mutex->the_lock == 1) {
        // Thread coming into waiting list
        // Thread must be placed behind other waiting threads
        if (mutex->waitlist_last_out != NULL) { // not the first in list
a0002c6a:       99 c3 18 00     ld.a %a3,[%a12]24
a0002c6e:       bc 36           jz.a %a3,a0002c7a <ccthread_mutex_lock+0x2a>
                        // connect with other threads
                        mutex->waitlist_last_out->synclist_next = my_tcb;
a0002c70:       89 32 90 09     st.a [%a3]16,%a2
                        my_tcb->synclist_prev = mutex->waitlist_last_out;
a0002c74:       4c c6           ld.w %d15,[%a12]24
a0002c76:       6c 23           st.w [%a2]12,%d15
                        // set myself as last thread in the whole list
                        mutex->waitlist_last_out = my_tcb;
a0002c78:       3c 03           j a0002c7e <ccthread_mutex_lock+0x2e>
                // waitlist_first_out is already set and stays the same
        }
        else { // thread is the first one in list
                mutex->waitlist_first_out = my_tcb;
a0002c7a:       89 c2 90 09     st.a [%a12]16,%a2
                mutex->waitlist_last_out = my_tcb;
a0002c7e:       89 c2 98 09     st.a [%a12]24,%a2
a0002c82:       0d 00           .hword 0x000d
a0002c84:       80 07           mov.d %d7,%a0
}
static inline void set_suspended(tcb_t *tcb) {
        tcb->sched_flags |= SF_SUSPENDED;
a0002c86:       19 20 10 10     ld.w %d0,[%a2]80
a0002c8a:       8f 20 40 01     or %d0,%d0,2
a0002c8e:       59 20 10 10     st.w [%a2]80,%d0
  static inline uint32_t cc_sl_unlock(cc_spinlock_t *lock) {
    uint32_t r = 0;
a0002c92:       82 0f           mov %d15,0
```

```
        #ifndef GCC
        __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
          : "memory");
        #else
        asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r)
          : "memory");
a0002c94:       49 ff 00 08     swap.w [%a15]0,%d15
a0002c98:       0d 00           .hword 0x000d
a0002c9a:       c0 07           .hword 0x07c0
a0002c9c:       d9 c2 04 00     lea %a2,[%a12]4
a0002ca0:       82 1f           mov %d15,1
a0002ca2:       49 2f 00 08     swap.w [%a2]0,%d15      // unlock(guard)
a0002ca6:       ee fd           jnz %d15,a0002ca0 <ccthread_mutex_lock+0x50>
        }
    // now go to sleep
    _tie();
    set_suspended(my_tcb);
    cc_sl_unlock(&mutex->guard);
    _untie();
    // after wakeup gain local spinlock
    cc_sl_lock(&mutex->guard);
    mutex->owner = th;
a0002ca8:       59 c8 08 00     st.w [%a12]8,%d8
        #ifndef GCC
        __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
          : "memory");
        #else
        asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r)
          : "memory");
a0002cac:       49 2f 00 08     swap.w [%a2]0,%d15      // lock(guard)
[fairmutex]
    cc_sl_unlock(&mutex->guard);
    return 0;
a0002cb0:       3c 17           j a0002cde <ccthread_mutex_lock+0x8e>
a0002cb2:       02 84           mov %d4,%d8
a0002cb4:       6d 00 12 02     call a00030d8 <get_thread4handler>
a0002cb8:       8f 88 20 f0     sha %d15,%d8,8
a0002cbc:       91 00 00 49     movh.a %a4,36864
a0002cc0:       60 f3           mov.a %a3,%d15
a0002cc2:       d9 44 00 40     lea %a4,[%a4]256
  <90000100 <__HEAP_MIN+0x8ff80100>>
a0002cc6:       54 c0           ld.w %d0,[%a12]
a0002cc8:       01 43 10 20     add.a %a2,%a3,%a4
a0002ccc:       df 10 cf 7f     jeq %d0,1,a0002c6a
  <ccthread_mutex_lock+0x1a>
  }
  // gained lock
  mutex->the_lock = 1;
a0002cd0:       82 1f           mov %d15,1
a0002cd2:       6c c0           st.w [%a12]0,%d15
  mutex->owner = th;
a0002cd4:       59 c8 08 00     st.w [%a12]8,%d8
  static inline uint32_t cc_sl_unlock(cc_spinlock_t *lock) {
    uint32_t r = 0;
```

```
a0002cd8:       82 0f           mov %d15,0
        #ifndef GCC
        __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
          : "memory");
        #else
        asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r)
          : "memory");
a0002cda:       49 ff 00 08     swap.w [%a15]0,%d15     // unlock_guard
[mutex_unlock]
  cc_sl_unlock(&mutex->guard);
  return 0;
}
a0002cde:       82 02           mov %d2,0
a0002ce0:       00 90           ret
a0002ce2:       00 00           nop
a0002ce4:       00 00           nop



<ccthread_mutex_unlock>:
a0002ce8 <ccthread_mutex_unlock>:
a0002ce8:       cd 6e 00 0e     mtcr $0xe006 (unknown SFR),%d14
int32_t ccthread_mutex_unlock(pthread_mutex_t* mutex) {
a0002cec:       40 4f           mov.aa %a15,%a4
        thread_handler th = get_current_thread_handler();
a0002cee:       6d 00 11 02     call a0003110 <get_current_thread_handler>
        if ((mutex->owner != th) && (th != SYSTEM_THREAD_HANDLER))
          return EPERM;
a0002cf2:       4c f2           ld.w %d15,[%a15]8
a0002cf4:       0b 2f 10 f1     ne %d15,%d15,%d2
a0002cf8:       8b 02 20 22     ne %d2,%d2,0
a0002cfc:       87 f2 00 20     and.t %d2,%d2,0,%d15,0
a0002d00:       82 1f           mov %d15,1
a0002d02:       6f 02 28 80     jnz.t %d2,0,a0002d52
  <ccthread_mutex_unlock+0x6a>
static inline uint32_t cc_sl_lock(cc_spinlock_t *lock) {
a0002d06:       d9 f2 04 00     lea %a2,[%a15]4
a0002d0a:       82 1f           mov %d15,1
a0002d0c:       49 2f 00 08     swap.w [%a2]0,%d15      // lock(guard)
[mutex_unlock]
a0002d10:       ee fd           jnz %d15,a0002d0a
  <ccthread_mutex_unlock+0x22>
        cc_sl_lock(&mutex->guard);
        if (mutex->waitlist_first_out != NULL) { //there are threads waiting
a0002d12:       c8 44           ld.a %a4,[%a15]16
a0002d14:       bd 04 18 00     jz.a %a4,a0002d44
  <ccthread_mutex_unlock+0x5c>
                tcb_t *tcb_to_unsuspend = mutex->waitlist_first_out;
                if (tcb_to_unsuspend->synclist_next != NULL) {
                  //there is more than one thread waiting
a0002d18:       99 43 10 00     ld.a %a3,[%a4]16
a0002d1c:       bc 36           jz.a %a3,a0002d28
  <ccthread_mutex_unlock+0x40>
                // set second thread in waitlist as "first out"
                mutex->waitlist_first_out = tcb_to_unsuspend->synclist_next;
a0002d1e:       e8 43           st.a [%a15]16,%a3
                  // remove connections to second thread
                        mutex->waitlist_first_out->synclist_prev = NULL;
a0002d20:       82 00           mov %d0,0
a0002d22:       59 30 0c 00     st.w [%a3]12,%d0
a0002d26:       3c 04           j a0002d2e <ccthread_mutex_unlock+0x46>
                }
                else { // there is only one thread waiting,
                       // so set list to NULL
                        mutex->waitlist_first_out = NULL;
                        mutex->waitlist_last_out = NULL;
a0002d28:       82 0f           mov %d15,0
a0002d2a:       68 6f           st.w [%a15]24,%d15
a0002d2c:       68 4f           st.w [%a15]16,%d15
static inline void set_suspended(tcb_t *tcb) {
        tcb->sched_flags |= SF_SUSPENDED;
}
static inline void unset_suspended(tcb_t *tcb) {
        tcb->sched_flags &= ~SF_SUSPENDED;
a0002d2e:       19 4f 10 10     ld.w %d15,[%a4]80
a0002d32:       8f 2f c0 f1     andn %d15,%d15,2
a0002d36:       59 4f 10 10     st.w [%a4]80,%d15
                }
        tcb_to_unsuspend->synclist_next = NULL;
a0002d3a:       82 00           mov %d0,0
a0002d3c:       59 40 10 00     st.w [%a4]16,%d0
  static inline uint32_t cc_sl_unlock(cc_spinlock_t *lock) {
    uint32_t r = 0;
a0002d40:       82 0f           mov %d15,0
        #ifndef GCC
        __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
          : "memory");
        #else
        asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r)
          : "memory");
a0002d42:       3c 05           j a0002d4c <ccthread_mutex_unlock+0x64>
                unset_suspended(tcb_to_unsuspend);
        cc_sl_unlock(&mutex->guard);
        return 0;
        }
        mutex->owner = NO_OWNER;
        mutex->the_lock = 0;
a0002d44:       82 00           mov %d0,0
a0002d46:       68 00           st.w [%a15]0,%d0
a0002d48:       82 f0           mov %d0,-1
a0002d4a:       68 20           st.w [%a15]8,%d0
        #ifndef GCC
        __asm("swap.w␣%0,[%2]0" : "=d" (r) : "0" (r), "a" ((uint32_t*) lock)
          : "memory");
        #else
        asm volatile ("swap.w␣[%1]0,␣%0": "=d" (r) : "a" (lock), "0" (r)
          : "memory");
a0002d4c:       49 2f 00 08     swap.w [%a2]0,%d15        // unlock(guard)
```

```
[mutex_unlock]
        cc_sl_unlock(&mutex->guard);
        return 0;
a0002d50:       82 0f           mov %d15,0
}
a0002d52:       02 f2           mov %d2,%d15
a0002d54:       00 90           ret


<set_suspended>:
sub.a %sp,8
st.a [%sp]4,%a4
ld.a %a2,[%sp]4
ld.a %a15,[%sp]4
ld.w %d15,[%a15]80
or %d15,2
st.w [%a2]80,%d15
ret

<unset_suspended>:
sub.a %sp,8
st.a [%sp]4,%a4
ld.a %a2,[%sp]4
ld.a %a15,[%sp]4
ld.w %d15,[%a15]80
andn %d15,%d15,2
st.w [%a2]80,%d15
ret

<_tie>:
.hword 0x000d
mov.d %d7,%a0
ret

<_untie>:
.hword 0x000d
.hword 0x07c0
ret

<get_current_thread_handler>:
sub.a %sp,8
call <get_current_tcb>
mov.aa %a15,%a2
st.a [%sp]4,%a15
ld.w %d15,[%sp]4
addih %d15,%d15,28672
add %d15,%d15,-256
sh %d15,-8
st.w [%sp]0,%d15
ld.w %d15,[%sp]0
mov %d2,%d15
ret
```

```
<get_thread4handler>:
sub.a %sp,16
st.w [%sp]12,%d4
ld.w %d15,[%sp]12
mov %d0,256
mul %d15,%d0
addih %d15,%d15,36864
addi %d15,%d15,256
st.w [%sp]8,%d15
ld.a %a15,[%sp]8
ld.w %d15,[%a15]4
and %d15,2
jnz %d15,a000526a <get_thread4handler+0x28>
mov.a %a15,0
add.a %a15,-1
st.a [%sp]4,%a15
j <get_thread4handler+0x2e>
ld.a %a15,[%sp]8
ld.w %d15,[%a15]8
st.w [%sp]4,%d15
ld.a %a2,[%sp]4 <a00b0002 <_SMALL_DATA_+0x27422>>
ret
```

Listing B.5:
Binary code: Semaphore with fetch-and-increment/fetch-and-decrement

```
  static inline int32_t pthread_sem_wait_suspend(pthread_sem_s_t *sem)
{
a0004534:         91 b0 00 2a      movh.a %a2,40971
a0004538:         d9 2f c0 f1      lea %a15,[%a2]8128 <a00b1fc0 <sem_s>>
a000453c:         1b 0a 00 f7      addi %d15,%d10,28672
a0004540:         49 ff 00 08      swap.w [%a15]0,%d15      // f&i(sem->value())
a0004544:         bb f0 ff 0f      mov.u %d0,65535
a0004548:         26 0f            and %d15,%d0
a000454a:         ee 4a            jnz %d15,a00045de <worker_col_sem_s+0xde>
        //suspending
        if((fetch_and_decrement(&sem->value)) <= 0)
        {
                  // sleep and enter waiting list; waking up from other thread
                  // calling sem_post


                  while(!fetch_and_decrement(&sem->waitlist_lock)){
a000454c:         1b 09 00 07      addi %d0,%d9,28672
a0004550:         02 0f            mov %d15,%d0
a0004552:         49 cf 00 08      swap.w [%a12]0,%d15
a0004556:         bb f0 ff 1f      mov.u %d1,65535
                  // lock (guard)
a000455a:         26 1f            and %d15,%d1
a000455c:         6e fa            jz %d15,a0004550 <worker_col_sem_s+0x50>
a000455e:         02 0f            mov %d15,%d0
a0004560:         49 ff 00 08      swap.w [%a15]0,%d15
a0004564:         bb f0 ff 0f      mov.u %d0,65535
a0004568:         26 0f            and %d15,%d0
a000456a:         ee 36            jnz %d15,a00045d6 <worker_col_sem_s+0xd6>
                  }

                  if((fetch_and_decrement(&sem->value)) <= 0)
                  {
                          thread_handler th = get_current_thread_handler();
a000456c:         6d ff c6 f5      call a00030f8 <get_current_thread_handler>
                          tcb_t *my_tcb = TH2TCB(th);
a0004570:         8f 82 20 20      sha %d2,%d2,8
a0004574:         91 00 00 f9      movh.a %a15,36864
a0004578:         60 23            mov.a %a3,%d2
a000457a:         d9 ff 00 40      lea %a15,[%a15]256 <90000100
                                   <__HEAP_MIN+0x8ff80100>>
a000457e:         01 f3 10 20      add.a %a2,%a3,%a15
a0004582:         91 b0 00 fa      movh.a %a15,40971
a0004586:         d9 ff d0 f1      lea %a15,[%a15]8144 <a00b1fd0 <sem_s+0x10>>
a000458a:         bb f0 ff ff      mov.u %d15,65535
a000458e:         49 ff 00 08      swap.w [%a15]0 <a00b0000
                                   <_SMALL_DATA_+0x27418>>,%d15
a0004592:         bb f0 ff 0f      mov.u %d0,65535
a0004596:         26 0f            and %d15,%d0
a0004598:         86 2f            sha %d15,2
a000459a:        91 b0 00 3a      movh.a %a3,40971
a000459e:        8b 8f 01 f0      add %d15,%d15,24
a00045a2:        d9 33 c0 f1      lea %a3,[%a3]8128 <a00b1fc0 <sem_s>>
a00045a6:        10 3f            addsc.a %a15,%a3,%d15,0
a00045a8:        e8 02            st.a [%a15]0,%a2
a00045aa:        bb f0 ff ff      mov.u %d15,65535
a00045ae:        49 df 00 08      swap.w [%a13]0,%d15
a00045b2:        0d 00            .hword 0x000d
a00045b4:        80 07            mov.d %d7,%a0
}
 */

static inline void set_suspended(tcb_t *tcb) {
        tcb->sched_flags |= SF_SUSPENDED;
a00045b6:        19 20 10 10      ld.w %d0,[%a2]80
a00045ba:        8f 20 40 01      or %d0,%d0,2
a00045be:        59 20 10 10      st.w [%a2]80,%d0
a00045c2:        bb f0 ff ff      mov.u %d15,65535
a00045c6:        49 cf 00 08      swap.w [%a12]0,%d15
a00045ca:        0d 00            .hword 0x000d
                 // unlock(guard)
a00045cc:        c0 07            .hword 0x07c0
a00045ce:        91 b0 00 ca      movh.a %a12,40971
a00045d2:        d9 cc cc f1      lea %a12,[%a12]8140 <a00b1fcc <sem_s+0xc>>
a00045d6:        bb f0 ff ff      mov.u %d15,65535
a00045da:        49 cf 00 08      swap.w [%a12]0 <a00b0000
                                  <_SMALL_DATA_+0x27418>>,%d15
                 {
                   pthread_sem_wait_suspend(&sem_s);
                   sem->waitlist_fifo[fetch_and_increment(&sem->fifo_next)]
                     = my_tcb;
                   fetch_and_increment(&sem->waitlist_entries);
                   _tie();
                   set_suspended(my_tcb);
                   fetch_and_increment(&sem->waitlist_lock);
                   _untie();
                   fetch_and_increment(&sem->waitlist_lock);
                   return 0;
                 }
                 else
                 {
                   fetch_and_increment(&sem->waitlist_lock);
                   return 0;
                 }
        }
        return 0;
}
```

Listing B.6:
Binary code: Subbarrier (including also the binary code of conditional variables)

```
/*
*** COPYRIGHT NOTICE FOR FUNCTION <barrier_wait> ***
*
* OpenSPARC T2 Processor File: barrier.c
* Copyright (c) 2006 Sun Microsystems, Inc.  All Rights Reserved.
* DO NOT ALTER OR REMOVE COPYRIGHT NOTICES.
*
* The above named program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public
* License version 2 as published by the Free Software Foundation.
*
* The above named program is distributed in the hope that it will be
* useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
* General Public License for more details.
*
* You should have received a copy of the GNU General Public
* License along with this work; if not, write to the Free Software
* Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.
*/


a0002a50 <barrier_wait>:
a0002a50:       cd 68 00 0e     mtcr $0xe006 (unknown SFR),%d8
        int barrier_wait(pthread_barrier_t *barrier)
        {
                struct _subbarrier *subbarrierp = barrier->subbarrierp;
a0002a54:       99 4f 1c 10     ld.a %a15,[%a4]92
a0002a58:       40 4c           mov.aa %a12,%a4
                pthread_mutex_lock(&subbarrierp->wait_lock);
a0002a5a:       d9 fd 0c 00     lea %a13,[%a15]12
a0002a5e:       40 d4           mov.aa %a4,%a13
a0002a60:       6d ff 4c ff     call a00028f8 <pthread_mutex_lock>
                if (subbarrierp->runners == 1)
                // last thread to reach barrier
a0002a64:       48 a0           ld.w %d0,[%a15]40
a0002a66:       df 10 17 80     jne %d0,1,a0002a94 <barrier_wait+0x44>
                {
                        if (barrier->maxcnt > 1)
                        // reset runner count and switch sub-barriers
a0002a6a:       54 c1           ld.w %d1,[%a12]
a0002a6c:       bf 21 21 00     jlt %d1,2,a0002aae <barrier_wait+0x5e>
                        {
                          subbarrierp->runners = barrier->maxcnt;
                          barrier->subbarrierp =
                          (barrier->subbarrierp == &barrier->subbarrier[0])
a0002a70:       80 c0           mov.d %d0,%a12
a0002a72:       9a 40           add %d15,%d0,4
a0002a74:       19 c0 1c 10     ld.w %d0,[%a12]92
a0002a78:       68 a1           st.w [%a15]40,%d1
a0002a7a:        80 c2            mov.d %d2,%a12
a0002a7c:        0b f0 00 01      eq %d0,%d0,%d15
                     ? &barrier->subbarrier[1] : &barrier->subbarrier[0];
                     // wake up waiting threads
                     pthread_cond_broadcast(&subbarrierp->wait_cond);
a0002a80:        40 f4            mov.aa %a4,%a15
a0002a82:        8b 02 03 10      add %d1,%d2,48
a0002a86:        2b f1 40 00      sel %d0,%d0,%d1,%d15
a0002a8a:        59 c0 1c 10      st.w [%a12]92,%d0
a0002a8e:        6d ff 4d ff      call a0002928 <pthread_cond_broadcast>
a0002a92:        3c 0e            j a0002aae <barrier_wait+0x5e>
                         }
                 }
                 else
                 {
                         subbarrierp->runners--; // one needed thread less
a0002a94:        c2 f0            add %d0,-1
a0002a96:        68 a0            st.w [%a15]40,%d0
                         while (subbarrierp->runners != barrier->maxcnt)
a0002a98:        4c c0            ld.w %d15,[%a12]0
a0002a9a:        5f f0 0a 00      jeq %d0,%d15,a0002aae <barrier_wait+0x5e>
                         {
                            pthread_cond_wait(&subbarrierp->wait_cond,
                            &subbarrierp->wait_lock);
  // wait until last thread to reach barrier
a0002a9e:        40 f4            mov.aa %a4,%a15
a0002aa0:        40 d5            mov.aa %a5,%a13
a0002aa2:        6d ff 63 ff      call a0002968 <pthread_cond_wait>
a0002aa6:        48 a0            ld.w %d0,[%a15]40
a0002aa8:        4c c0            ld.w %d15,[%a12]0
a0002aaa:        5f f0 fa ff      jne %d0,%d15,a0002a9e <barrier_wait+0x4e>
                         }
                 }
                 pthread_mutex_unlock(&subbarrierp->wait_lock);
a0002aae:        40 d4            mov.aa %a4,%a13
a0002ab0:        6d ff 34 ff      call a0002918 <pthread_mutex_unlock>
                 return 0;
         }
a0002ab4:        82 02            mov %d2,0
a0002ab6:        00 90            ret




a0002928 <pthread_cond_broadcast>:
a0002928:        cd 0a 00 0e      mtcr $dpm0_0,%d10
int pthread_cond_broadcast(pthread_cond_t *cond){
        return ccthread_cond_broadcast(cond);
a000292c:        6d 00 5a 01      call a0002be0 <ccthread_cond_broadcast>
a0002930:        00 90            ret
```

```
a0002be0 <ccthread_cond_broadcast>:
a0002be0:        cd 38 00 0e      mtcr $dpm0_3,%d8
int32_t ccthread_cond_broadcast(pthread_cond_t* cond) {
        if (cond == NULL) return EINVAL;
a0002be4:        3b 60 01 20      mov %d2,22
a0002be8:        bd 04 17 00      jz.a %a4,a0002c16
                                  <ccthread_cond_broadcast+0x36>
        tcb_t *tcb = cond->waitlist_first_out;
a0002bec:        99 42 04 00      ld.a %a2,[%a4]4
        tcb_t *tcb_tmp;
        while (tcb != NULL) {
a0002bf0:        bc 2f            jz.a %a2,a0002c0e
                                  <ccthread_cond_broadcast+0x2e>
static inline void set_suspended(tcb_t *tcb) {
        tcb->sched_flags |= SF_SUSPENDED; }
static inline void unset_suspended(tcb_t *tcb) {
        tcb->sched_flags &= ~SF_SUSPENDED;
a0002bf2:        19 2f 10 10      ld.w %d15,[%a2]80
a0002bf6:        8f 2f c0 f1      andn %d15,%d15,2
a0002bfa:        40 2f            mov.aa %a15,%a2
a0002bfc:        59 2f 10 10      st.w [%a2]80,%d15
                 unset_suspended(tcb);
                 tcb_tmp = tcb;
                 tcb = tcb_tmp->synclist_next;
                 tcb_tmp->synclist_next = NULL;
                 tcb_tmp->synclist_prev = NULL;
a0002c00:        82 0f            mov %d15,0
a0002c02:        99 22 10 00      ld.a %a2,[%a2]16
a0002c06:        68 3f            st.w [%a15]12,%d15
a0002c08:        68 4f            st.w [%a15]16,%d15
a0002c0a:        bd 02 f4 ff      jnz.a %a2,a0002bf2
                                  <ccthread_cond_broadcast+0x12>
        }
          cond->waitlist_first_out = NULL;
          cond->waitlist_last_out = NULL;
a0002c0e:        82 0f            mov %d15,0
a0002c10:        6c 42            st.w [%a4]8,%d15
a0002c12:        6c 41            st.w [%a4]4,%d15
        return E_OK;
}
a0002c14:        82 02            mov %d2,0
a0002c16:        00 90            ret


a0002968 <pthread_cond_wait>:
a0002968:        cd 0a 00 0e      mtcr $dpm0_0,%d10
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex){
        return ccthread_cond_wait(cond, mutex);
a000296c:        6d 00 d6 00      call a0002b18 <ccthread_cond_wait>
a0002970:        00 90            ret
```

```
a0002b18 <ccthread_cond_wait>:
a0002b18:       cd 80 00 0e     mtcr $0xe008 (unknown SFR),%d0
    return EBUSY;
  return E_OK;
}
int32_t ccthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex) {
a0002b1c:       40 5c           mov.aa %a12,%a5
  if (cond == NULL)
a0002b1e:       3b 60 01 20     mov %d2,22
a0002b22:       bd 04 3a 00     jz.a %a4,a0002b96 <ccthread_cond_wait+0x7e>
    return EINVAL;
  if ((cond->mutex != NULL) && (cond->mutex != mutex))
a0002b26:       cc 40           ld.a %a15,[%a4]0
a0002b28:       bc f3           jz.a %a15,a0002b2e
                                <ccthread_cond_wait+0x16>
a0002b2a:       7d 5f 36 80     jne.a %a15,%a5,a0002b96
                                <ccthread_cond_wait+0x7e>
static inline tcb_t *get_current_tcb(void) {
        return (tcb_t *) MSS_MY_TCB;
a0002b2e:       85 9f 08 08     ld.a %a15,90000008 <__HEAP_MIN+0x8ff80008>
    return EINVAL;
  tcb_t *my_tcb = get_current_tcb();
  thread_handler th = TCB2TH(my_tcb);
a0002b32:       7b 00 00 27     movh %d2,28672
a0002b36:       80 f1           mov.d %d1,%a15
a0002b38:       1b 02 f0 2f     addi %d2,%d2,-256
  if (mutex->owner != th)
a0002b3c:       19 c0 08 00     ld.w %d0,[%a12]8
a0002b40:       1a 21           add %d15,%d1,%d2
a0002b42:       06 8f           sh %d15,-8
a0002b44:       82 12           mov %d2,1
a0002b46:       5f f0 28 80     jne %d0,%d15,a0002b96
                                <ccthread_cond_wait+0x7e>
    return EPERM;
  cond->mutex = mutex;
a0002b4a:       f4 4c           st.a [%a4],%a12
  // Thread coming into waiting list
  // Thread must be placed behind other waiting threads
  if (cond->waitlist_last_out != NULL) { // not the first in list
a0002b4c:       99 42 08 00     ld.a %a2,[%a4]8
a0002b50:       bc 27           jz.a %a2,a0002b5e <ccthread_cond_wait+0x46>
        // connect with other threads
        cond->waitlist_last_out->synclist_next = my_tcb;
a0002b52:       ec 24           st.a [%a2]16,%a15
        my_tcb->synclist_prev = cond->waitlist_last_out;
a0002b54:       19 42 08 00     ld.w %d2,[%a4]8
a0002b58:       68 32           st.w [%a15]12,%d2
        // set myself as last thread in the whole list
        cond->waitlist_last_out = my_tcb;
a0002b5a:       ec 42           st.a [%a4]8,%a15
a0002b5c:       3c 03           j a0002b62 <ccthread_cond_wait+0x4a>
        // waitlist_first_out is already set and stays the same
  }
  else { // coming HRT thread is the first one in list
                cond->waitlist_first_out = my_tcb;
                cond->waitlist_last_out = my_tcb;
a0002b5e:       ec 42           st.a [%a4]8,%a15
a0002b60:       ec 41           st.a [%a4]4,%a15
a0002b62:       0d 00           .hword 0x000d
a0002b64:       80 07           mov.d %d7,%a0
  }
  _tie();
  ccthread_mutex_unlock(mutex);
a0002b66:       40 c4           mov.aa %a4,%a12
a0002b68:       6d 00 b8 00     call a0002cd8 <ccthread_mutex_unlock>
        tcb->sched_flags &= ~SF_SWAPOUT;
}
static inline void set_suspended(tcb_t *tcb) {
a0002b6c:       8f 8f 20 f0     sha %d15,%d15,8
a0002b70:       91 00 00 39     movh.a %a3,36864
a0002b74:       60 f2           mov.a %a2,%d15
a0002b76:       d9 33 00 40     lea %a3,[%a3]256
                                <90000100 <__HEAP_MIN+0x8ff80100>>
a0002b7a:       01 32 10 f0     add.a %a15,%a2,%a3
        tcb->sched_flags |= SF_SUSPENDED;
a0002b7e:       19 f0 10 10     ld.w %d0,[%a15]80
a0002b82:       8f 20 40 01     or %d0,%d0,2
a0002b86:       59 f0 10 10     st.w [%a15]80,%d0
a0002b8a:       0d 00           .hword 0x000d
a0002b8c:       c0 07           .hword 0x07c0
  set_suspended(TH2TCB(th));
  _untie();
  ccthread_mutex_lock(mutex);
a0002b8e:       40 c4           mov.aa %a4,%a12
a0002b90:       6d 00 58 00     call a0002c40 <ccthread_mutex_lock>
  return E_OK;
a0002b94:       82 02           mov %d2,0
}
a0002b96:       00 90           ret
```

LISTING B.7:
Binary code: Barriers with fetch-and-increment

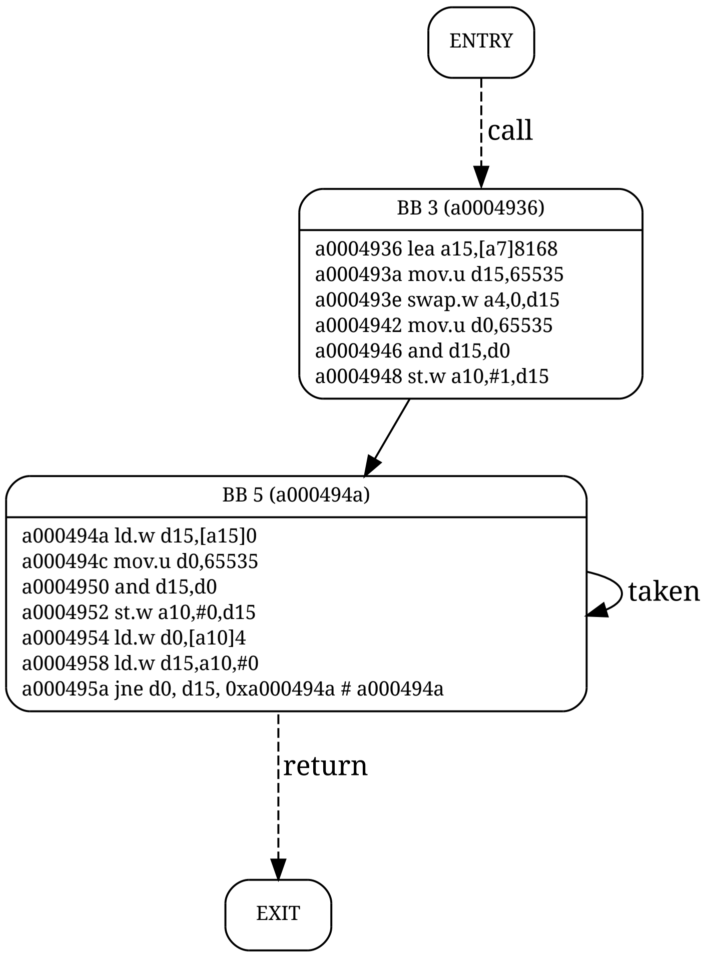
```
a0002a50 <barrier_wait>:
a0002a50:       cd a4 00 0e     mtcr $0xe00a (unknown SFR),%d4
int barrier_wait(pthread_barrier_t *barrier)
{
a0002a54:       40 4d           mov.aa %a13,%a4
a0002a56:       d9 4c 08 00     lea %a12,[%a4]8
a0002a5a:       7b 00 00 07     movh %d0,28672
        #endif
        return data & 0x0000FFFF;
  }
  static inline uint32_t fetch_and_decrement(uint32_t *address) {
        uint32_t data = 0x70007000;
a0002a5e:       1b 00 00 f7     addi %d15,%d0,28672
        #ifndef GCC
        __asm("swap.w␣%0,[%2]0" : "=d" (data) : "0" (data),
          "a" ((uint32_t*) address) : "memory");
        #else
        asm volatile ("swap.w␣[%2]0,␣%0" : "=d" (data): "0" (data),
          "a" (address) : "memory");
a0002a62:       49 cf 00 08     swap.w [%a12]0,%d15
a0002a66:       bb f0 ff 1f     mov.u %d1,65535
a0002a6a:       26 1f           and %d15,%d1
a0002a6c:       6e f9           jz %d15,a0002a5e <barrier_wait+0xe>
a0002a6e:       d9 df 04 00     lea %a15,[%a13]4
a0002a72:       bb f0 ff ff     mov.u %d15,65535
a0002a76:       49 ff 00 08     swap.w [%a15]0,%d15
        while(!fetch_and_decrement(&barrier->waitlist_lock))
        {       }
        uint32_t cur_runner = fetch_and_increment(&barrier->runners);
        uint32_t needed = (uint32_t) barrier->needed;
a0002a7a:       54 d0           ld.w %d0,[%a13]
a0002a7c:       26 1f           and %d15,%d1
        if( cur_runner >= (needed - 1))//last needed thread to reach barrier
a0002a7e:       c2 f0           add %d0,-1
a0002a80:       3f 0f 1a 80     jlt.u %d15,%d0,a0002ab4 <barrier_wait+0x64>
        {
                uint32_t i;
                for(i = 0; i < needed-1; i++)
a0002a84:       bf 10 13 80     jlt.u %d0,1,a0002aaa <barrier_wait+0x5a>
a0002a88:       60 04           mov.a %a4,%d0
a0002a8a:       d9 43 ff ff     lea %a3,[%a4]-1
a0002a8e:       3b c0 00 00     mov %d0,12
        {
        tcb_t *tcb_to_unsuspend = (tcb_t*)barrier->waitlist_fifo[i];
a0002a92:       60 0f           mov.a %a15,%d0
a0002a94:       01 fd 10 f0     add.a %a15,%a13,%a15
a0002a98:       c8 02           ld.a %a2,[%a15]0
static inline void set_suspended(tcb_t *tcb) {
        tcb->sched_flags |= SF_SUSPENDED;
}
static inline void unset_suspended(tcb_t *tcb) {
        tcb->sched_flags &= ~SF_SUSPENDED;
a0002a9a:       19 2f 10 10     ld.w %d15,[%a2]80
a0002a9e:       8f 2f c0 f1     andn %d15,%d15,2
a0002aa2:       59 2f 10 10     st.w [%a2]80,%d15
a0002aa6:       c2 40           add %d0,4
a0002aa8:       fc 35           loop %a3,a0002a92 <barrier_wait+0x42>
a0002aaa:       bb f0 ff ff     mov.u %d15,65535
a0002aae:       49 cf 00 08     swap.w [%a12]0,%d15
a0002ab2:       3c 1f           j a0002af0 <barrier_wait+0xa0>
                        unset_suspended(tcb_to_unsuspend);
                }
                fetch_and_increment(&barrier->waitlist_lock);
        }
        else
        {
                thread_handler th = get_current_thread_handler();
a0002ab4:       6d 00 2e 03     call a0003110 <get_current_thread_handler>
                tcb_t *my_tcb = TH2TCB(th);
a0002ab8:       8f 82 20 20     sha %d2,%d2,8
a0002abc:       91 00 00 49     movh.a %a4,36864
                barrier->waitlist_fifo[cur_runner] = my_tcb;
a0002ac0:       86 2f           sha %d15,2
a0002ac2:       60 23           mov.a %a3,%d2
a0002ac4:       8b cf 00 f0     add %d15,%d15,12
a0002ac8:       10 d2           addsc.a %a2,%a13,%d15,0
a0002aca:       d9 44 00 40     lea %a4,[%a4]256
    <90000100 <__HEAP_MIN+0x8ff80100>>
a0002ace:       01 43 10 f0     add.a %a15,%a3,%a4
a0002ad2:       ec 20           st.a [%a2]0,%a15
a0002ad4:       0d 00           .hword 0x000d
a0002ad6:       80 07           mov.d %d7,%a0
}
static inline void set_suspended(tcb_t *tcb) {
        tcb->sched_flags |= SF_SUSPENDED;
a0002ad8:       19 f0 10 10     ld.w %d0,[%a15]80
a0002adc:       8f 20 40 01     or %d0,%d0,2
a0002ae0:       59 f0 10 10     st.w [%a15]80,%d0
a0002ae4:       bb f0 ff ff     mov.u %d15,65535
a0002ae8:       49 cf 00 08     swap.w [%a12]0,%d15
a0002aec:       0d 00           .hword 0x000d
a0002aee:       c0 07           .hword 0x07c0
                _tie();
                set_suspended(my_tcb);
                fetch_and_increment(&barrier->waitlist_lock);
                _untie();
        }
        return 0;
}
a0002af0:       82 02           mov %d2,0
a0002af2:       00 90           ret
```

C Appendix: CFGs of SW Synchronisations

C.1. CFGs of Software Synchronisations

In this appendix the CFGs derived from the static WCET tool OTAWA, which are used as baseline for the static timing analyses in Chapter 4, are presented.

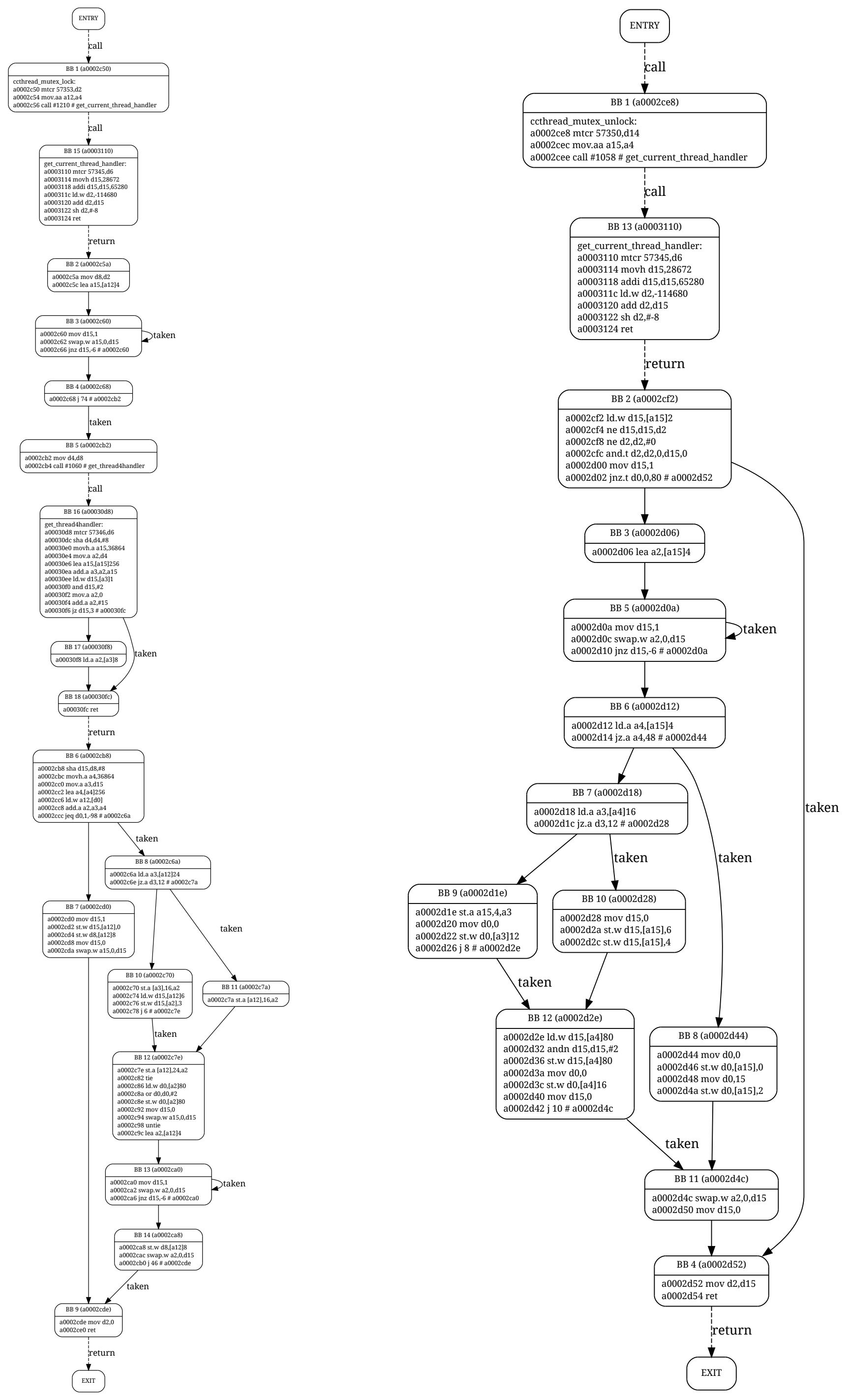
The CFGs of spin locks with TAS and spin locks with F&I/F&D are shown as part of the mutex lock CFG. For example BB3 and BB13 of Figure C.2(a) and BB5 of Figure C.2(b) show the control flow of a TAS spin lock. BB8 of Figure C.2(b) presents the control flow of the TAS spin unlock. BB8 of Figure C.3(a) shows the F&D spin lock control flow, and e.g. BB14 of Figure C.3(a) the F&I unlock control flow of a F&I/F&D spin lock. The ticket unlock CFG is not shown here, as the instructions in the binary are very similar to the F&D unlock CFG.



```
BB 3 (a0004936)
a0004936 lea a15,[a7]8168
a000493a mov.u d15,65535
a000493e swap.w a4,0,d15
a0004942 mov.u d0,65535
a0004946 and d15,d0
a0004948 st.w a10,#1,d15
```

```
BB 5 (a000494a)
a000494a ld.w d15,[a15]0
a000494c mov.u d0,65535
a0004950 and d15,d0
a0004952 st.w a10,#0,d15
a0004954 ld.w d0,[a10]4
a0004958 ld.w d15,a10,#0
a000495a jne d0, d15, 0xa000494a # a000494a
```

FIGURE C.1.: OTAWA Ouput: CFG of ticket lock function with F&I/F&D



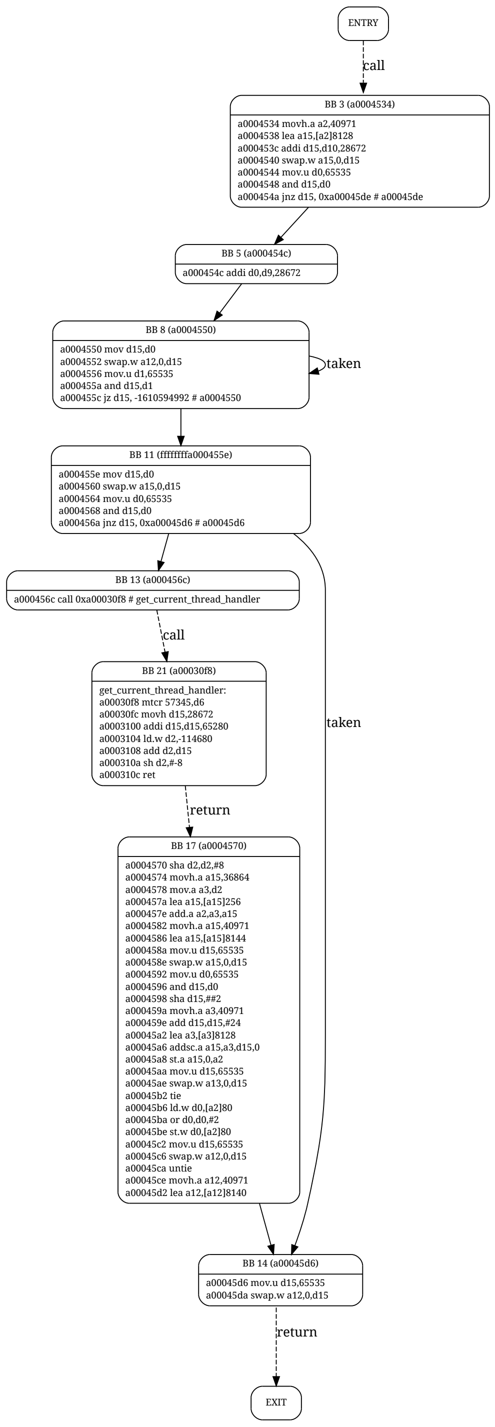
```
BB 1 (a0002c50)
ccthread_mutex_lock:
a0002c50 mtcr 57353,d2
a0002c54 mov.aa a12,a4
a0002c56 call #1210 # get_current_thread_handler

BB 15 (a0003110)
get_current_thread_handler:
a0003110 mtcr 57345,d6
a0003114 movh d15,28672
a0003118 addi d15,d15,65280
a000311c ld.w d2,-114680
a0003120 add d2,d15
a0003122 sh d2,#-8
a0003124 ret

BB 2 (a0002c5a)
a0002c5a mov d8,d2
a0002c5c lea a15,[a12]4

BB 3 (a0002c60)
a0002c60 mov d15,1
a0002c62 swap.w a15,0,d15
a0002c66 jnz d15,-6 # a0002c60

BB 4 (a0002c68)
a0002c68 j 74 # a0002cb2

BB 5 (a0002cb2)
a0002cb2 mov d4,d8
a0002cb4 call #1060 # get_thread4handler

BB 16 (a00030d8)
get_thread4handler:
a00030d8 mtcr 57346,d6
a00030dc sha d4,d4,#8
a00030e0 movh.a a15,36864
a00030e4 mov.a a2,d4
a00030e6 lea a15,[a15]256
a00030ea add.a a3,a2,a15
a00030ee ld.w d15,[a3]1
a00030f0 and d15,#2
a00030f2 mov.a a2,0
a00030f4 add.a a2,#15
a00030f6 jz d15,3 # a00030fc

BB 17 (a00030f8)
a00030f8 ld.a a2,[a3]8

BB 18 (a00030fc)
a00030fc ret

BB 6 (a0002cb8)
a0002cb8 sha d15,d8,#8
a0002cbc movh.a a4,36864
a0002cc0 mov.a a3,d15
a0002cc2 lea a4,[a4]256
a0002cc6 ld.w a12,[d0]
a0002cc8 add.a a2,a3,a4
a0002ccc jeq d0,1,-98 # a0002c6a

BB 8 (a0002c6a)
a0002c6a ld.a a3,[a12]24
a0002c6e jz.a d3,12 # a0002c7a

BB 7 (a0002cd0)
a0002cd0 mov d15,1
a0002cd2 st.w d15,[a12],0
a0002cd4 st.w d8,[a12]8
a0002cd8 mov d15,0
a0002cda swap.w a15,0,d15

BB 10 (a0002c70)
a0002c70 st.a [a3],16,a2
a0002c74 ld.w d15,[a12]6
a0002c76 st.w d15,[a2],3
a0002c78 j 6 # a0002c7e

BB 11 (a0002c7a)
a0002c7a st.a [a12],16,a2

BB 12 (a0002c7e)
a0002c7e st.a [a12],24,a2
a0002c82 tie
a0002c86 ld.w d0,[a2]80
a0002c8a or d0,d0,#2
a0002c8e st.w d0,[a2]80
a0002c92 mov d15,0
a0002c94 swap.w a15,0,d15
a0002c98 untie
a0002c9c lea a2,[a12]4

BB 13 (a0002ca0)
a0002ca0 mov d15,1
a0002ca2 swap.w a2,0,d15
a0002ca6 jnz d15,-6 # a0002ca0

BB 14 (a0002ca8)
a0002ca8 st.w d8,[a12]8
a0002cac swap.w a2,0,d15
a0002cb0 j 46 # a0002cde

BB 9 (a0002cde)
a0002cde mov d2,0
a0002ce0 ret
```

(a)

```
BB 1 (a0002ce8)
ccthread_mutex_unlock:
a0002ce8 mtcr 57350,d14
a0002cec mov.aa a15,a4
a0002cee call #1058 # get_current_thread_handler

BB 13 (a0003110)
get_current_thread_handler:
a0003110 mtcr 57345,d6
a0003114 movh d15,28672
a0003118 addi d15,d15,65280
a000311c ld.w d2,-114680
a0003120 add d2,d15
a0003122 sh d2,#-8
a0003124 ret

BB 2 (a0002cf2)
a0002cf2 ld.w d15,[a15]2
a0002cf4 ne d15,d15,d2
a0002cf8 ne d2,d2,#0
a0002cfc and.t d2,d2,0,d15,0
a0002d00 mov d15,1
a0002d02 jnz.t d0,0,80 # a0002d52

BB 3 (a0002d06)
a0002d06 lea a2,[a15]4

BB 5 (a0002d0a)
a0002d0a mov d15,1
a0002d0c swap.w a2,0,d15
a0002d10 jnz d15,-6 # a0002d0a

BB 6 (a0002d12)
a0002d12 ld.a a4,[a15]4
a0002d14 jz.a a4,48 # a0002d44

BB 7 (a0002d18)
a0002d18 ld.a a3,[a4]16
a0002d1c jz.a d3,12 # a0002d28

BB 9 (a0002d1e)
a0002d1e st.a a15,4,a3
a0002d20 mov d0,0
a0002d22 st.w d0,[a3]12
a0002d26 j 8 # a0002d2e

BB 10 (a0002d28)
a0002d28 mov d15,0
a0002d2a st.w d15,[a15],6
a0002d2c st.w d15,[a15],4

BB 12 (a0002d2e)
a0002d2e ld.w d15,[a4]80
a0002d32 andn d15,d15,#2
a0002d36 st.w d15,[a4]80
a0002d3a mov d0,0
a0002d3c st.w d0,[a4]16
a0002d40 mov d15,0
a0002d42 j 10 # a0002d4c

BB 8 (a0002d44)
a0002d44 mov d0,0
a0002d46 st.w d0,[a15],0
a0002d48 mov d0,15
a0002d4a st.w d0,[a15],2

BB 11 (a0002d4c)
a0002d4c swap.w a2,0,d15
a0002d50 mov d15,0

BB 4 (a0002d52)
a0002d52 mov d2,d15
a0002d54 ret
```

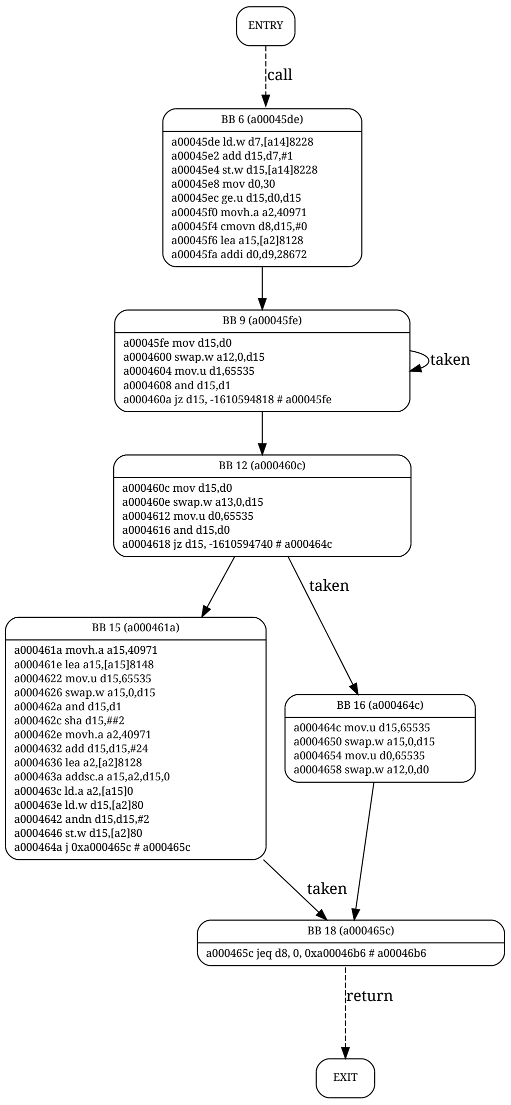
(b)

FIGURE C.2.: OTAWA Ouput: CFG of (fair) *mutex lock* function (a) and (fair) *mutex unlock* function (b) from matmul program



```
BB 3 (a0004534)
a0004534 movh.a a2,40971
a0004538 lea a15,[a2]8128
a000453c addi d15,d10,28672
a0004540 swap.w a15,0,d15
a0004544 mov.u d0,65535
a0004548 and d15,d0
a000454a jnz d15, 0xa00045de # a00045de

BB 5 (a000454c)
a000454c addi d0,d9,28672

BB 8 (a0004550)
a0004550 mov d15,d0
a0004552 swap.w a12,0,d15
a0004556 mov.u d1,65535
a000455a and d15,d1
a000455c jz d15, -1610594992 # a0004550

BB 11 (ffffffffa000455e)
a000455e mov d15,d0
a0004560 swap.w a15,0,d15
a0004564 mov.u d0,65535
a0004568 and d15,d0
a000456a jnz d15, 0xa00045d6 # a00045d6

BB 13 (a000456c)
a000456c call 0xa00030f8 # get_current_thread_handler

BB 21 (a00030f8)
get_current_thread_handler:
a00030f8 mtcr 57345,d6
a00030fc movh d15,28672
a0003100 addi d15,d15,65280
a0003104 ld.w d2,-114680
a0003108 add d2,d15
a000310a sh d2,#-8
a000310c ret

BB 17 (a0004570)
a0004570 sha d2,d2,#8
a0004574 movh.a a15,36864
a0004578 mov.a a3,d2
a000457a lea a15,[a15]256
a000457e add.a a2,a3,a15
a0004582 movh.a a15,40971
a0004586 lea a15,[a15]8144
a000458a mov.u d15,65535
a000458e swap.w a15,0,d15
a0004592 mov.u d0,65535
a0004596 and d15,d0
a0004598 sha d15,##2
a000459a movh.a a3,40971
a000459e add d15,d15,#24
a00045a2 lea a3,[a3]8128
a00045a6 addsc.a a15,a3,d15,0
a00045a8 st.a a15,0,a2
a00045aa mov.u d15,65535
a00045ae swap.w a13,0,d15
a00045b2 tie
a00045b6 ld.w d0,[a2]80
a00045ba or d0,d0,#2
a00045be st.w d0,[a2]80
a00045c2 mov.u d15,65535
a00045c6 swap.w a12,0,d15
a00045ca untie
a00045ce movh.a a12,40971
a00045d2 lea a12,[a12]8140

BB 14 (a00045d6)
a00045d6 mov.u d15,65535
a00045da swap.w a12,0,d15
```

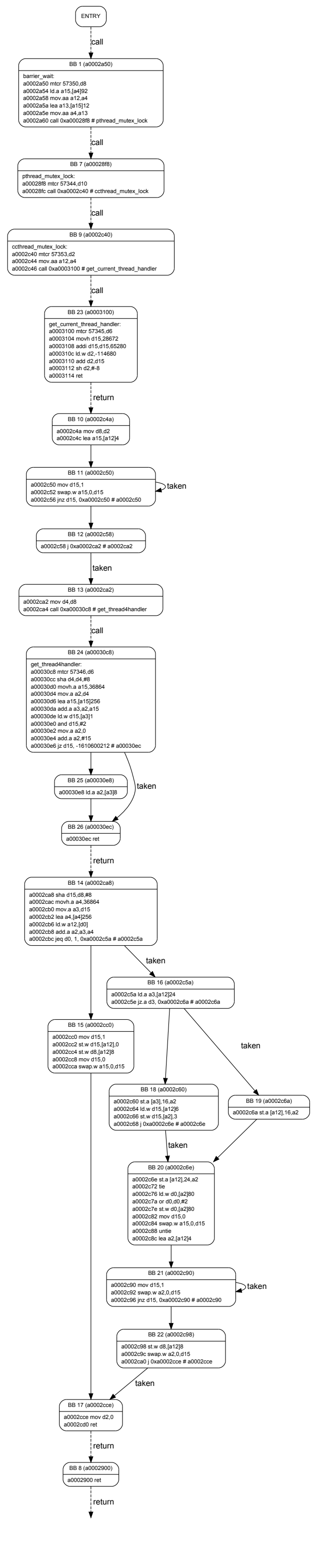
(a)



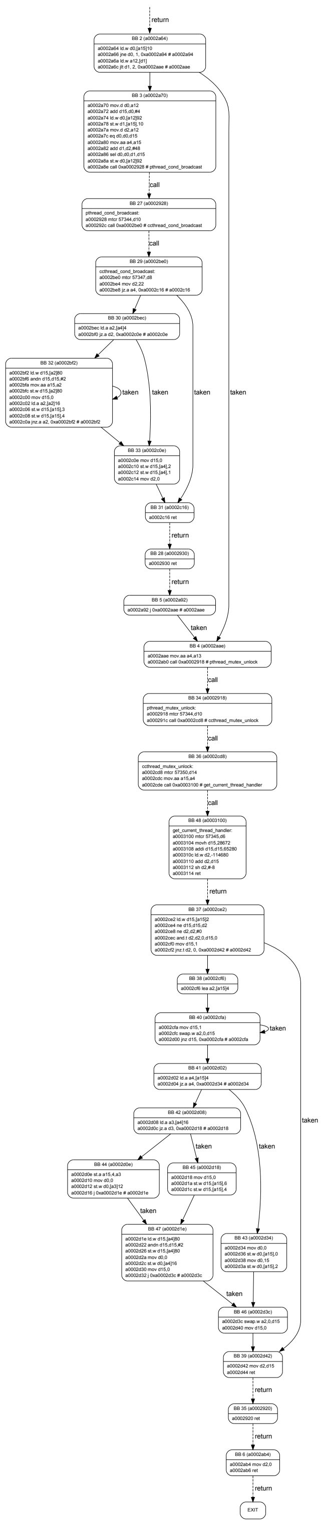
```
BB 6 (a00045de)
a00045de ld.w d7,[a14]8228
a00045e2 add d15,d7,#1
a00045e4 st.w d15,[a14]8228
a00045e8 mov d0,30
a00045ec ge.u d15,d0,d15
a00045f0 movh.a a2,40971
a00045f4 cmovn d8,d15,#0
a00045f6 lea a15,[a2]8128
a00045fa addi d0,d9,28672

BB 9 (a00045fe)
a00045fe mov d15,d0
a0004600 swap.w a12,0,d15
a0004604 mov.u d1,65535
a0004608 and d15,d1
a000460a jz d15, -1610594818 # a00045fe

BB 12 (a000460c)
a000460c mov d15,d0
a000460e swap.w a13,0,d15
a0004612 mov.u d0,65535
a0004616 and d15,d0
a0004618 jz d15, -1610594740 # a000464c

BB 15 (a000461a)
a000461a movh.a a15,40971
a000461e lea a15,[a15]8148
a0004622 mov.u d15,65535
a0004626 swap.w a15,0,d15
a000462a and d15,d1
a000462c sha d15,##2
a000462e movh.a a2,40971
a0004632 add d15,d15,#24
a0004636 lea a2,[a2]8128
a000463a addsc.a a15,a2,d15,0
a000463c ld.a a2,[a15]0
a000463e ld.w d15,[a2]80
a0004642 andn d15,d15,#2
a0004646 st.w d15,[a2]80
a000464a j 0xa000465c # a000465c

BB 16 (a000464c)
a000464c mov.u d15,65535
a0004650 swap.w a15,0,d15
a0004654 mov.u d0,65535
a0004658 swap.w a12,0,d0

BB 18 (a000465c)
a000465c jeq d8, 0, 0xa00046b6 # a00046b6
```

(b)

FIGURE C.3.: OTAWA Ouput: CFG of Binary Semaphores with F&I/F&D. *sem_wait()* in (a) and *sem_post()* in (b).

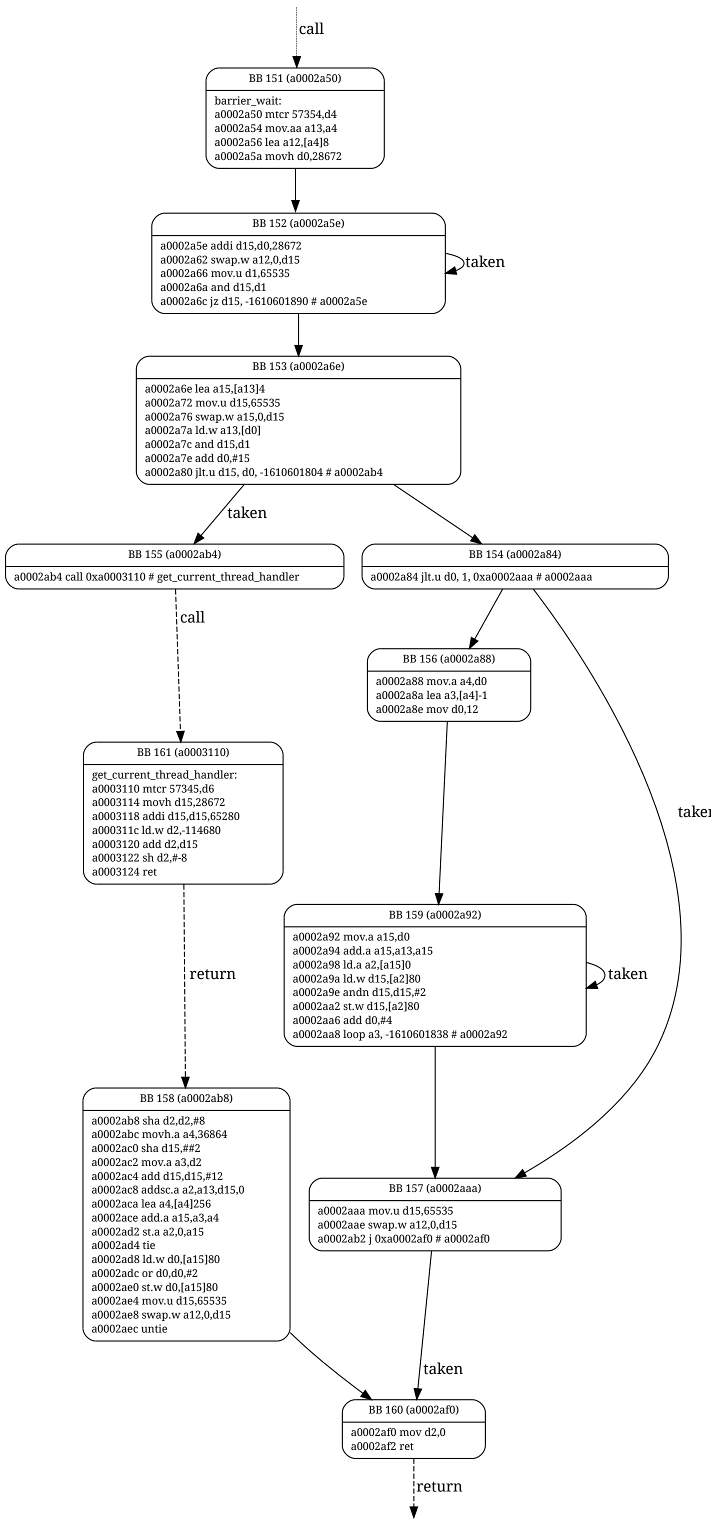


```
BB 1 (a0002a50)
barrier_wait:
a0002a50 mtcr 57350,d8
a0002a54 ld.a a15,[a4]92
a0002a58 mov.aa a12,a4
a0002a5a lea a13,[a15]12
a0002a5e mov.aa a4,a13
a0002a60 call 0xa00028f8 # pthread_mutex_lock

BB 7 (a00028f8)
pthread_mutex_lock:
a00028f8 mtcr 57344,d10
a00028fc call 0xa0002c40 # ccthread_mutex_lock

BB 9 (a0002c40)
ccthread_mutex_lock:
a0002c40 mtcr 57353,d2
a0002c44 mov.aa a12,a4
a0002c46 call 0xa0003100 # get_current_thread_handler

BB 23 (a0003100)
get_current_thread_handler:
a0003100 mtcr 57345,d6
a0003104 movh d15,28672
a0003108 addi d15,d15,65280
a000310c ld.w d2,-114680
a0003110 add d2,d15
a0003112 sh d2,#-8
a0003114 ret

BB 10 (a0002c4a)
a0002c4a mov d8,d2
a0002c4c lea a15,[a12]4

BB 11 (a0002c50)
a0002c50 mov d15,1
a0002c52 swap.w a15,0,d15
a0002c56 jnz d15, 0xa0002c50 # a0002c50

BB 12 (a0002c58)
a0002c58 j 0xa0002ca2 # a0002ca2

BB 13 (a0002ca2)
a0002ca2 mov d4,d8
a0002ca4 call 0xa00030c8 # get_thread4handler

BB 24 (a00030c8)
get_thread4handler:
a00030c8 mtcr 57346,d6
a00030cc sha d4,d4,#8
a00030d0 movh.a a15,36864
a00030d4 mov.a a2,d4
a00030d6 lea a15,[a15]256
a00030da add.a a3,a2,a15
a00030de ld.w d15,[a3]1
a00030e0 and d15,#2
a00030e2 mov.a a2,0
a00030e4 add.a a2,#15
a00030e6 jz d15, -1610600212 # a00030ec

BB 25 (a00030e8)
a00030e8 ld.a a2,[a3]8

BB 26 (a00030ec)
a00030ec ret

BB 14 (a0002ca8)
a0002ca8 sha d15,d8,#8
a0002cac movh.a a4,36864
a0002cb0 mov.a a3,d15
a0002cb2 lea a4,[a4]256
a0002cb6 ld.w a12,[d0]
a0002cb8 add.a a2,a3,a4
a0002cbc jeq d0, 1, 0xa0002c5a # a0002c5a

BB 16 (a0002c5a)
a0002c5a ld.a a3,[a12]24
a0002c5e jz.a d3, 0xa0002c6a # a0002c6a

BB 15 (a0002cc0)
a0002cc0 mov d15,1
a0002cc2 st.w d15,[a12],0
a0002cc4 st.w d8,[a12]8
a0002cc8 mov d15,0
a0002cca swap.w a15,0,d15

BB 18 (a0002c60)
a0002c60 st.a [a3],16,a2
a0002c64 ld.w d15,[a12]6
a0002c66 st.w d15,[a2],3
a0002c68 j 0xa0002c6e # a0002c6e

BB 19 (a0002c6a)
a0002c6a st.a [a12],16,a2

BB 20 (a0002c6e)
a0002c6e st.a [a12],24,a2
a0002c72 tie
a0002c76 ld.w d0,[a2]80
a0002c7a or d0,d0,#2
a0002c7e st.w d0,[a2]80
a0002c82 mov d15,0
a0002c84 swap.w a15,0,d15
a0002c88 untie
a0002c8c lea a2,[a12]4

BB 21 (a0002c90)
a0002c90 mov d15,1
a0002c92 swap.w a2,0,d15
a0002c96 jnz d15, 0xa0002c90 # a0002c90

BB 22 (a0002c98)
a0002c98 st.w d8,[a12]8
a0002c9c swap.w a2,0,d15
a0002ca0 j 0xa0002cce # a0002cce

BB 17 (a0002cce)
a0002cce mov d2,0
a0002cd0 ret

BB 8 (a0002900)
a0002900 ret
```



```
BB 2 (a0002a64)
a0002a64 ld.w d0,[a15]10
a0002a66 jne d0, 1, 0xa0002a94 # a0002a94
a0002a6a ld.w a12,[d1]
a0002a6c jlt d1, 2, 0xa0002aae # a0002aae

BB 3 (a0002a70)
a0002a70 mov.d d0,a12
a0002a72 add d15,d0,#4
a0002a74 ld.w d0,[a12]92
a0002a78 st.w d1,[a15],10
a0002a7a mov.d d2,a12
a0002a7c eq d0,d0,d15
a0002a80 mov.aa a4,a15
a0002a82 add d1,d2,#48
a0002a86 sel d0,d0,d1,d15
a0002a8a st.w d0,[a12]92
a0002a8e call 0xa0002928 # pthread_cond_broadcast

BB 27 (a0002928)
pthread_cond_broadcast:
a0002928 mtcr 57344,d10
a000292c call 0xa0002be0 # ccthread_cond_broadcast

BB 29 (a0002be0)
ccthread_cond_broadcast:
a0002be0 mtcr 57347,d8
a0002be4 mov d2,22
a0002be8 jz.a a4, 0xa0002c16 # a0002c16

BB 30 (a0002bec)
a0002bec ld.a a2,[a4]4
a0002bf0 jz.a d2, 0xa0002c0e # a0002c0e

BB 32 (a0002bf2)
a0002bf2 ld.w d15,[a2]80
a0002bf6 andn d15,d15,#2
a0002bfa mov.aa a15,a2
a0002bfc st.w d15,[a2]80
a0002c00 mov d15,0
a0002c02 ld.a a2,[a2]16
a0002c06 st.w d15,[a15],3
a0002c08 st.w d15,[a15],4
a0002c0a jnz.a a2, 0xa0002bf2 # a0002bf2

BB 33 (a0002c0e)
a0002c0e mov d15,0
a0002c10 st.w d15,[a4],2
a0002c12 st.w d15,[a4],1
a0002c14 mov d2,0

BB 31 (a0002c16)
a0002c16 ret

BB 28 (a0002930)
a0002930 ret

BB 5 (a0002a92)
a0002a92 j 0xa0002aae # a0002aae

BB 4 (a0002aae)
a0002aae mov.aa a4,a13
a0002ab0 call 0xa0002918 # pthread_mutex_unlock

BB 34 (a0002918)
pthread_mutex_unlock:
a0002918 mtcr 57344,d10
a000291c call 0xa0002cd8 # ccthread_mutex_unlock

BB 36 (a0002cd8)
ccthread_mutex_unlock:
a0002cd8 mtcr 57350,d14
a0002cdc mov.aa a15,a4
a0002cde call 0xa0003100 # get_current_thread_handler

BB 48 (a0003100)
get_current_thread_handler:
a0003100 mtcr 57345,d6
a0003104 movh d15,28672
a0003108 addi d15,d15,65280
a000310c ld.w d2,-114680
a0003110 add d2,d15
a0003112 sh d2,#-8
a0003114 ret

BB 37 (a0002ce2)
a0002ce2 ld.w d15,[a15]2
a0002ce4 ne d15,d15,d2
a0002ce8 ne d2,d2,#0
a0002cec and.t d2,d2,0,d15,0
a0002cf0 mov d15,1
a0002cf2 jnz.t d2, 0, 0xa0002d42 # a0002d42

BB 38 (a0002cf6)
a0002cf6 lea a2,[a15]4

BB 40 (a0002cfa)
a0002cfa mov d15,1
a0002cfc swap.w a2,0,d15
a0002d00 jnz d15, 0xa0002cfa # a0002cfa

BB 41 (a0002d02)
a0002d02 ld.a a4,[a15]4
a0002d04 jz.a a4, 0xa0002d34 # a0002d34

BB 42 (a0002d08)
a0002d08 ld.a a3,[a4]16
a0002d0c jz.a d3, 0xa0002d18 # a0002d18

BB 44 (a0002d0e)
a0002d0e st.a a15,4,a3
a0002d10 mov d0,0
a0002d12 st.w d0,[a3]12
a0002d16 j 0xa0002d1e # a0002d1e

BB 45 (a0002d18)
a0002d18 mov d15,0
a0002d1a st.w d15,[a15],6
a0002d1c st.w d15,[a15],4

BB 47 (a0002d1e)
a0002d1e ld.w d15,[a4]80
a0002d22 andn d15,d15,#2
a0002d26 st.w d15,[a4]80
a0002d2a mov d0,0
a0002d2c st.w d0,[a4]16
a0002d30 mov d15,0
a0002d32 j 0xa0002d3c # a0002d3c

BB 43 (a0002d34)
a0002d34 mov d0,0
a0002d36 st.w d0,[a15],0
a0002d38 mov d0,15
a0002d3a st.w d0,[a15],2

BB 46 (a0002d3c)
a0002d3c swap.w a2,0,d15
a0002d40 mov d15,0

BB 39 (a0002d42)
a0002d42 mov d2,d15
a0002d44 ret

BB 35 (a0002920)
a0002920 ret

BB 6 (a0002ab4)
a0002ab4 mov d2,0
a0002ab6 ret

EXIT
```

FIGURE C.3.: OTAWA Ouput: CFG of Subbariers



```
BB 151 (a0002a50)
barrier_wait:
a0002a50 mtcr 57354,d4
a0002a54 mov.aa a13,a4
a0002a56 lea a12,[a4]8
a0002a5a movh d0,28672

BB 152 (a0002a5e)
a0002a5e addi d15,d0,28672
a0002a62 swap.w a12,0,d15
a0002a66 mov.u d1,65535
a0002a6a and d15,d1
a0002a6c jz d15, -1610601890 # a0002a5e

BB 153 (a0002a6e)
a0002a6e lea a15,[a13]4
a0002a72 mov.u d15,65535
a0002a76 swap.w a15,0,d15
a0002a7a ld.w a13,[d0]
a0002a7c and d15,d1
a0002a7e add d0,#15
a0002a80 jlt.u d15, d0, -1610601804 # a0002ab4

BB 155 (a0002ab4)
a0002ab4 call 0xa0003110 # get_current_thread_handler

BB 154 (a0002a84)
a0002a84 jlt.u d0, 1, 0xa0002aaa # a0002aaa

BB 156 (a0002a88)
a0002a88 mov.a a4,d0
a0002a8a lea a3,[a4]-1
a0002a8e mov d0,12

BB 161 (a0003110)
get_current_thread_handler:
a0003110 mtcr 57345,d6
a0003114 movh d15,28672
a0003118 addi d15,d15,65280
a000311c ld.w d2,-114680
a0003120 add d2,d15
a0003122 sh d2,#-8
a0003124 ret

BB 159 (a0002a92)
a0002a92 mov.a a15,d0
a0002a94 add.a a15,a13,a15
a0002a98 ld.a a2,[a15]0
a0002a9a ld.w d15,[a2]80
a0002a9e andn d15,d15,#2
a0002aa2 st.w d15,[a2]80
a0002aa6 add d0,#4
a0002aa8 loop a3, -1610601838 # a0002a92

BB 158 (a0002ab8)
a0002ab8 sha d2,d2,#8
a0002abc movh.a a4,36864
a0002ac0 sha d15,##2
a0002ac2 mov.a a3,d2
a0002ac4 add d15,d15,#12
a0002ac8 addsc.a a2,a13,d15,0
a0002aca lea a4,[a4]256
a0002ace add.a a15,a3,a4
a0002ad2 st.a a2,0,a15
a0002ad4 tie
a0002ad8 ld.w d0,[a15]80
a0002adc or d0,d0,#2
a0002ae0 st.w d0,[a15]80
a0002ae4 mov.u d15,65535
a0002ae8 swap.w a12,0,d15
a0002aec untie

BB 157 (a0002aaa)
a0002aaa mov.u d15,65535
a0002aae swap.w a12,0,d15
a0002ab2 j 0xa0002af0 # a0002af0

BB 160 (a0002af0)
a0002af0 mov d2,0
a0002af2 ret
```

FIGURE C.4.: OTAWA Ouput: CFG of F&I barriers

Bibliography

[Abdelzaher et al. 2004] Abdelzaher, Tarek F. ; Sharma, Vivek ; Lu, Chenyang: A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling. In: *IEEE Transactions on Computers* 53 (2004), March, No. 3, p. 334–350. – DOI http://dx.doi.org/10.1109/TC.2004.1261839. – ISSN 0018-9340 (page 21)

[Adve and Hill 1990] Adve, Sarita V. ; Hill, Mark D.: Weak Ordering - A New Definition. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1990 (ISCA '90), p. 2–14. – DOI http://doi.acm.org/10.1145/325164.325100. – ISBN 0-89791-366-3 (page 8, 89, 90, 97, 101, 133)

[Akesson and Goossens 2011] Akesson, Benny ; Goossens, Kees: Architectures and Modeling of Predictable Memory Controllers for Improved System Integration. In: *Proceedings of Design, Automation and Test in Europe*. Leuven, Belgium : European Design and Automation Association, March 2011 (DATE '11), p. 851–856. – ISSN 1530-1591 (page 100)

[Akesson et al. 2007] Akesson, Benny ; Goossens, Kees ; Ringhofer, Markus: Predator: A Predictable SDRAM Memory Controller. In: *Proceedings of the 5th IEEE/ACM International Conference on HW/SW Codesign and System Synthesis*. New York, NY, USA : ACM, 2007 (CODES+ISSS '07), p. 251–256. – DOI http://doi.acm.org/10.1145/1289816.1289877. – ISBN 978-1-59593-824-4 (page 2, 19, 25)

[Alexander et al. 1977] Alexander, Christopher ; Ishikawa, Sara ; Silverstein, Murray ; Jacobson, Max ; Fiksdahl-King, Ingrid ; Angel, Shlomo: *A Pattern Language - Towns, Buildings, Construction*. New York, NY, US : Oxford University Press, 1977. – ISBN 0-19-501919-9, 978-0-19-501919-3 (page 116)

[Almasi and Gottlieb 1989] Almasi, George S. ; Gottlieb, Allan: *Highly Parallel Computing*. Redwood City, CA, USA : Benjamin-Cummings Publishing Co., Inc., 1989. – ISBN 0-8053-0177-1 (page 24)

[Anderson et al. 1997] Anderson, James H. ; Ramamurthy, Srikanth ; Jeffay, Kevin: Real-Time Computing with Lock-Free Shared Objects. In: *ACM Transactions on Computer Systems (TOCS)* 15 (1997), May, No. 2, p. 134–165. – DOI http://doi.acm.org/10.1145/253145.253159. – ISSN 0734-2071 (page 47, 49)

[Anderson 1990] Anderson, Thomas E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. In: *IEEE Transactions on Parallel Distributed Systems* 1 (1990), January, No. 1, p. 6–16. – DOI http://dx.doi.org/10.1109/71.80120. – ISSN 1045-9219 (page 13, 44)

[Andrei et al. 2008] Andrei, Alexandru ; Eles, Petru ; Peng, Zebo ; Rosen, Jakob: Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: *Proceedings of the 21st International Conference on VLSI Design.* Washington, DC, USA : IEEE Computer Society, 2008 (VLSID '08), p. 103–110. – DOI http://dx.doi.org/10.1109/VLSI.2008.33. – ISBN 0-7695-3083-4 (page 86)

[ARM11 MPCore] ARM Ltd.: *ARM11 MPCore Processor, Technical Reference Manual, Revision: r2p0.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360f/index.html. – Last Retrieved: April 2013 (page 19)

[ARMv6-M ISA 2010] ARM Ltd.: *ARMv6 Instruction Set Architecture.* http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html. September 2010. – Last Retrieved: April 2013 (page 18, 31, 100)

[ARMv7-M ISA 2010] ARM Ltd.: *ARMv7 Instruction Set Architecture.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403c/index.html. February 2010. – Last Retrieved: April 2013 (page 16, 18, 100, 133)

[Arvind et al. 1989] Arvind ; Nikhil, Rishiyur S. ; Pingali, Keshav K.: I-Structures: Data Structures for Parallel Computing. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11 (1989), October, No. 4, p. 598–632. – DOI http://doi.acm.org/10.1145/69558.69562. – ISSN 0164-0925 (page 18, 100)

[Asanovic et al. 2009] Asanovic, Krste ; Bodik, Rastislav ; Demmel, James ; Keaveny, Tony ; Keutzer, Kurt ; Kubiatowicz, John ; Morgan, Nelson ; Patterson, David ; Sen, Koushik ; Wawrzynek, John ; Wessel, David ; Yelick, Katherine: A View of the Parallel Computing Landscape. In: *Commun. ACM* 52 (2009), October, No. 10, p. 56–67. – DOI http://doi.acm.org/10.1145/1562764.1562783. – ISSN 0001-0782 (page 117, 121, 138)

[Austin et al. 2002] Austin, Todd ; Larson, Eric ; Ernst, Dan: SimpleScalar: An Infrastructure for Computer System Modeling. In: *IEEE Computer* 35 (2002), February, No. 2, p. 59–67. – DOI http://dx.doi.org/10.1109/2.982917. – ISSN 0018-9162 (page 86)

[Ballabriga et al. 2010] Ballabriga, Clément ; Cassé, Hugues ; Rochange, Christine ; Sainrat, Pascal: OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: Min, Sang (Pub.) ; Pettit, Robert (Pub.) ; Puschner, Peter (Pub.) ; Ungerer, Theo (Pub.): *Proceedings of the 8th IFIP WG 10.2 International Conference on Software technologies for Embedded and Ubiquitous Systems.* Berlin / Heidelberg, Germany : Springer-Verlag, 2010 (SEUS '10), p. 35–46. – DOI http://dl.acm.org/citation.cfm?id=1927882.1927891. – ISBN 3-642-16255-X, 978-3-642-16255-8 (page 7, 55)

[Baroni et al. 2003] Baroni, Aline L. ; Guéhéneuc, Yann-Gaël ; Albin-Amiot, Hervé: Design Patterns Formalization / Ecole Nationale Supérieure des Techniques

Industrielles et des Mines de Nantes. June 2003 (No. 03/3/INFO). – Technical Report (page 116)

[Barros and Pinho 2011] Barros, António ; Pinho, Luis M.: Software Transactional Memory as a Building Block for Parallel Embedded Real-Time Systems. In: *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. Washington, DC, USA : IEEE Computer Society, September 2011 (SEAA '11), p. 251–255. – DOI http://dx.doi.org/10.1109/SEAA.2011.46 (page 50, 51)

[Barth et al. 1991] Barth, Paul S. ; Nikhil, Rishiyur S. ; Arvind: M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In: *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. London, UK : Springer-Verlag, 1991, p. 538–568. – DOI http://dl.acm.org/citation.cfm?id=645420.652538. – ISBN 3-540-54396-1 (page 18)

[Baruah et al. 2010] Baruah, Sanjoy K. ; Li, Haohan ; Stougie, Leen: Towards the Design of Certifiable Mixed-criticality Systems. In: *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA : IEEE Computer Society, 2010 (RTAS '10), p. 13–22. – DOI http://dx.doi.org/10.1109/RTAS.2010.10. – ISBN 978-0-7695-4001-6 (page 1)

[Beck and Cunningham 1987] Beck, Kent ; Cunningham, Ward: Using Pattern Languages for Object-Oriented Programs. 1987 (No. CR-87-43). – Technical Report (page 116)

[Blieberger 2002] Blieberger, Johann: Data-Flow Frameworks for Worst-Case Execution Time Analysis. In: *Real-Time Syst.* 22 (2002), May, No. 3, p. 183–227. – DOI http://dx.doi.org/10.1023/A:1014535317056. – ISSN 0922-6443 (page 87, 88)

[Bollella et al. 2000] Bollella, Gregory ; Gosling, James ; Brosgol, Benjamin M.: *The Real-Time Specification for Java*. Amsterdam, The Netherlands : Addison-Wesley Longman, 2000. – ISBN 0201703238, 978-0201703238 (page 50)

[Bonenfant et al. 2010] Bonenfant, Armelle ; Broster, Ian ; Ballabriga, Clément ; Bernat, Guillem ; Cassé, Hugues ; Houston, Michael ; Merriam, Nicholas ; Michiel, Marianne de ; Rochange, Christine ; Sainrat, Pascal: Coding guidelines for WCET analysis using measurement-based and static analysis techniques / IRIT, Université Paul Sabatier, Toulouse. March 2010 (IRIT/RR–2010-8–FR). – Technical Report (page 54)

[Boniol et al. 2012] Boniol, Frédéric ; Cassé, Hugues ; Noulard, Eric ; Pagetti, Claire: Deterministic execution model on COTS hardware. In: *Proceedings of the 25th International Conference on Architecture of Computing Systems*. Berlin / Heidelberg, Germany : Springer-Verlag, 2012 (ARCS '12), p. 98–110. – DOI http://dx.doi.org/10.1007/978-3-642-28293-5_9. – ISBN 978-3-642-28292-8 (page 2, 19, 22)

[Brandenburg et al. 2008] Brandenburg, Björn B. ; Calandrino, John M. ; Block, Aaron ; Leontyev, Hennadiy ; Anderson, James H.: Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In: *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium.* Los Alamitos, CA, USA : IEEE Computer Society Press, 2008 (RTAS '08), p. 342–353. – DOI http://dx.doi.org/10.1109/RTAS.2008.27. – ISSN 1545-3421 (page 1, 51, 88)

[Brause 2004] Brause, Rüdiger W.: *Betriebssysteme - Grundlagen und Konzepte.* 3. Berlin / Heidelberg, Germany : Springer-Verlag, 2004. – ISBN 3642185312 (page 5)

[Brooks 1986] Brooks, Eugene D.: The Butterfly Barrier. In: *International Journal on Parallel Programming* 15 (1986), October, No. 4, p. 295–307. – DOI http://dx.doi.org/10.1007/BF01407877. – ISSN 0885-7458 (page 46)

[Brooks 1975] Brooks, Frederick P.: *The mythical man-month : essays on software engineering.* Boston, MA, USA : Addison-Wesley, 1975. – ISBN 0201006502 (page 117)

[Brooks 1995] Brooks, Frederick P.: *The mythical man-month : essays on software engineering.* Anniversary Edition. Boston, MA, USA : Addison-Wesley, 1995. – ISBN 0201835959 (page 117)

[Bull and Ball 2003] Bull, J. M. ; Ball, Carwyn: Point-to-Point Synchronisation on Shared Memory Architectures. 2003. – Technical Report. http://www.compunity.org/events/ewomp03/omptalks/Monday/Session2/T10p.pdf (page 10)

[Buschmann et al. 2007] Buschmann, Frank ; Henney, Kevlin ; Schmidt, Douglas C.: *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages.* Volume 5, 1st Edition. West Sussex, GB : John Wiley & Sons, Inc., 2007. – ISBN 0471486485, 978-0-471-48648-0 (page 117)

[Buschmann et al. 1996] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-Oriented Software Architecture: A System of Patterns.* Volume 1, 1st Edition. West Sussex, GB : John Wiley & Sons, Inc., 1996. – ISBN 0471958697, 9780471958697 (page 116)

[Buttazzo 2004] Buttazzo, Giorgio C.: *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Santa Clara, CA, USA : Springer-Verlag TELOS, 2004. – ISBN 0387231374 (page 1, 5)

[Cachopo and Rito-Silva 2006] Cachopo, João ; Rito-Silva, António: Versioned Boxes as the Basis for Memory Transactions. In: *Sci. Comput. Program.* 63 (2006), December, No. 2, p. 172–185. – DOI http://dx.doi.org/10.1016/j.scico.2006.05.009. – ISSN 0167-6423 (page 51)

[Carriero and Gelernter 1989] Carriero, Nicholas ; Gelernter, David: How to Write Parallel Programs: A Guide to the Perplexed. In: *ACM Comput. Surv.* 21 (1989), September, No. 3, p. 323–357. – DOI http://doi.acm.org/10.1145/72551.72553. – ISSN 0360-0300 (page 117)

[Chattopadhyay et al. 2012] Chattopadhyay, Sudipta ; Kee, Chong L. ; Roychoudhury, Abhik ; Kelter, Timon ; Marwedel, Peter ; Falk, Heiko: A Unified WCET Analysis Framework for Multi-core Platforms. In: *Proceedings of the IEE Real-Time and Embedded Technology and Applications Symposium* (2012), p. 99–108. – ISSN 1080-1812 (page 1, 2, 86)

[Chattopadhyay et al. 2010] Chattopadhyay, Sudipta ; Roychoudhury, Abhik ; Mitra, Tulika: Modeling shared cache and bus in multi-cores for timing analysis. In: *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*. New York, NY, USA : ACM, 2010 (SCOPES '10), p. 6:1–6:10. – DOI http://doi.acm.org/10.1145/1811212.1811220. – ISBN 978-1-4503-0084-1 (page 86)

[Chen and Tripathi 1994] Chen, Chia-Mei ; Tripathi, Satish K.: Multiprocessor Priority Ceiling Based Protocols. College Park, MD, USA : University of Maryland at College Park, 1994 (No. UMIACS-TR-94-42). – Technical Report (page 47)

[Chen et al. 1994] Chen, Chia-Mei ; Tripathi, Satish K. ; Blackmore, Alex: A Resource Synchronization Protocol for Multiprocessor Real-Time Systems. In: *Proceedings of the 1994 International Conference on Parallel Processing - Volume 03*. Washington, DC, USA : IEEE Computer Society, 1994 (ICPP '94), p. 159–162. – DOI http://dx.doi.org/10.1109/ICPP.1994.44. – ISBN 0-8493-2493-9 (page 47)

[Chen and Lin 1990] Chen, Min-Ih ; Lin, Kwei-Jay: Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. In: *Real-Time Syst.* 2 (1990), October, No. 4, p. 325–346. – DOI http://dx.doi.org/10.1007/BF01995676. – ISSN 0922-6443 (page 47, 50)

[Chen and Lin 1991] Chen, Min-Ih ; Lin, Kwei-Jay: A Priority Ceiling Protocol for Multiple-Instance Resources. In: *Proceedings of the Twelfth Real-Time Systems Symposium*. Los Alamitos, CA, USA : IEEE Computer Society Press, December 1991, p. 140–149. – DOI http://dx.doi.org/10.1109/REAL.1991.160367 (page 47)

[Colin and Petters 2003] Colin, Antoine ; Petters, Stefan M.: Experimental Evaluation of Code Properties for WCET Analysis. In: *Proceedings of the 24th IEEE International Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2003 (RTSS '03), p. 190–. – DOI http://dl.acm.org/citation.cfm?id=956418.956585. – ISBN 0-7695-2044-8 (page 57)

[Coplien and Schmidt 1995] Coplien, James (Pub.) ; Schmidt, Douglas C. (Pub.): *Pattern Languages of Program Design*. Boston, MA, USA : Addison-Wesley, 1995. – ISBN 0201607344 (page 116)

[Coplien 1992] Coplien, James O.: *Advanced C++ Programming Styles and Idioms*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1992. – ISBN 0-201-54855-0 (page 116)

[Cordes and Marwedel 2012] Cordes, Daniel ; Marwedel, Peter: Multi-Objective Aware Extraction of Task-Level Parallelism Using Genetic Algorithms. In: *Proceedings of Design, Automation and Test in Europe.* Leuven, Belgium : European Design and Automation Association, March 2012 (DATE '12), p. 394–399 (page 131, 132)

[Cordes et al. 2010] Cordes, Daniel ; Marwedel, Peter ; Mallik, Arindam: Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming. In: *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/software codesign and system synthesis.* New York, NY, USA : ACM, 2010 (CODES/ISSS '10), p. 267–276. – DOI http://doi.acm.org/10.1145/1878961.1879009. – ISBN 978-1-60558-905-3 (page 131, 132)

[Craig 1993] Craig, Travis S.: Queuing Spin Lock Algorithms to Support Timing Predictability. In: *Real-Time Systems Symposium 1993*, DOI http://dx.doi.org/10.1109/REAL.1993.393505, December 1993, p. 148–157 (page 13, 45)

[Culler et al. 1993] Culler, David E. ; Dusseau, Andrea ; Goldstein, Seth C. ; Krishnamurthy, Arvind ; Lumetta, Steven S. ; Eicken, Thorsten H. von ; Yelick, Katherine A.: Parallel Programming in Split-C. In: *Proceedings of Supercomputing '93*, DOI http://dx.doi.org/10.1109/SUPERC.1993.1263470, November 1993, p. 262–273. – ISSN 1063-9535 (page 100)

[Culler et al. 1997] Culler, David E. ; Gupta, Anoop ; Singh, J. P.: *Parallel Computer Architecture: A Hardware/Software Approach.* 1st Edition. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. – ISBN 1-55860-343-3 (page 7, 9, 10, 15)

[Culler et al. 1991] Culler, David E. ; Sah, Anurag ; Schauser, Klaus E. ; Eicken, Thorsten von ; Wawrzynek, John: Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In: *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* New York, NY, USA : ACM, 1991 (ASPLOS-IV), p. 164–175. – DOI http://doi.acm.org/10.1145/106972.106990. – ISBN 0-89791-380-9 (page 100)

[Cullmann et al. 2010] Cullmann, Christoph ; Ferdinand, Christian ; Gebhard, Gernot ; Grund, Daniel ; Maiza, Claire ; Reineke, Jan ; Triquet, Benoît ; Wilhelm, Reinhard: Predictability Considerations in the Design of Multi-Core Embedded Systems. In: *Proceedings of Embedded Real Time Software and Systems*, May 2010, p. 36–42 (page 2, 86, 89, 100)

[d'Ausbourg et al. 2012] d'Ausbourg, Bruno ; Boyer, Marc ; Noulard, Eric ; Pagetti, Claire: Deterministic Execution on Many-Core Platforms: application to the SCC. In: *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium.* Potsdam, Germany : Universitätsverlag Potsdam, 2012, p. 43–48. – ISBN 978-3-86956-169-1 (page 2, 22)

[Dechev and Stroustrup 2009] Dechev, Damian ; Stroustrup, Björn: Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software. In: *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology.* Washington, DC, USA : IEEE Computer Society, 2009 (SMC-IT '09), p. 323–330. – DOI http://dx.doi.org/10.1109/SMC-IT.2009.45. – ISBN 978-0-7695-3637-8 (page 47, 49)

[Devietti et al. 2010] Devietti, Joseph ; Lucia, Brandon ; Ceze, Luis ; Oskin, Mark: DMP: Deterministic Shared-Memory Multiprocessing. In: *Micro, IEEE* 30 (2010), No. 1, p. 40–49. – DOI http://dx.doi.org10.1109/MM.2010.14. – ISSN 0272-1732 (page 101)

[Devietti et al. 2011] Devietti, Joseph ; Nelson, Jacob ; Bergan, Tom ; Ceze, Luis ; Grossman, Dan: RCDC: A Relaxed Consistency Deterministic Computer. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* New York, NY, USA : ACM, 2011 (ASPLOS XVI), p. 67–78. – DOI http://doi.acm.org/10.1145/1950365.1950376. – ISBN 978-1-4503-0266-1 (page 101)

[Dijkstra 1965] Dijkstra, Edsger W.: Solution of a Problem in Concurrent Programming Control. In: *Commun. ACM* 8 (1965), September, p. 569–. – DOI http://doi.acm.org/10.1145/365559.365617. – ISSN 0001-0782 (page 8, 9)

[Dijkstra 1968] Dijkstra, Edsger W.: Cooperating Sequential Processes. In: Genuys, F. (Pub.): *Programming Languages: NATO Advanced Study Institute.* Academic Press, 1968, p. 43–112 (page 13, 38)

[Douglass 2006] Douglass, Bruce P.: *Real-time design patterns.* 6th Edition. Boston, MA, USA : Addison-Wesley, 2006. – ISBN 0-201-69956-7, 978-0-201-69956-2 (page 116, 131, 132)

[Dubois et al. 1986] Dubois, Michel ; Scheurich, Christoph ; Briggs, Faye A.: Memory Access Buffering in Multiprocessors. In: *Proceedings of 13th Annual International Symposium on Computer Architecture* Vol. 14. Los Alamitos, CA, USA : IEEE Computer Society Press, June 1986, p. 434–442. – DOI http://dl.acm.org/citation.cfm?id=17407.17406. – ISBN 0-8186-0719-X (page 8, 23)

[Edwards and Lee 2007] Edwards, Stephen A. ; Lee, Edward A.: The case for the precision timed (PRET) machine. In: *Proceedings of the 44th Annual Design Automation Conference.* New York, NY, USA : ACM, 2007 (DAC '07), p. 264–265. – DOI http://doi.acm.org/10.1145/1278480.1278545. – ISBN 978-1-59593-627-1 (page 2, 87)

[Engdahl and Chung 2010] Engdahl, Jonathan R. ; Chung, Dukki: Lock-Free Data Structure for Multi-core Processors. In: *Proceedings of International Conference on Control Automation and Systems*, October 2010 (ICCAS '10), p. 984–989 (page 31, 49)

[Eser 2010] Eser, Arthur: *Evaluierung paralleler Anwendungen auf dem MERASA Prozessor*, University of Augsburg, Master thesis, September 2010. – http://www.informatik.uni-augsburg.de/lehrstuehle/sik/publikationen/finished_thesises/201009_eser/ (page 83, 84)

[Fahmy et al. 2009] Fahmy, Sherif F. ; Ravindran, Binoy ; Jensen, E. D.: Response Time Analysis of Software Transactional Memory-Based Distributed Real-Time Systems. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA : ACM, 2009 (SAC '09), p. 334–338. – DOI http://doi.acm.org/10.1145/1529282.1529353. – ISBN 978-1-60558-166-8 (page 50, 51)

[Fahmy et al. 2008] Fahmy, Sherif F. ; Ravindran, Binoy ; Jensen, E. D.: On Scalable Synchronization for Distributed Embedded Real-Time Systems. In: *Proceedings of the 6th International Workshop on Software Technologies for Embedded and Ubiquitous Systems*. Berlin / Heidelberg, Germany : Springer-Verlag, 2008 (SEUS '08), p. 394–405. – DOI http://dx.doi.org/10.1007/978-3-540-87785-1_35. – ISBN 978-3-540-87784-4 (page 51)

[Fan et al. 2012] Fan, Dongrui ; Zhang, Hao ; Wang, Da ; Ye, Xiaochun ; Song, Fenglong ; Li, Guojie ; Sun, Ninghui: Godson-T: An Efficient Many-Core Processor Exploring Thread-Level Parallelism. In: *Micro, IEEE* 32 (2012), No. 2, p. 38–47. – DOI http://dx.doi.org/10.1109/MM.2012.32. – ISSN 0272-1732 (page 17)

[Felber et al. 2008] Felber, Pascal ; Fetzer, Christof ; Riegel, Torvald: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : ACM, 2008 (PPoPP '08), p. 237–246. – DOI http://doi.acm.org/10.1145/1345206.1345241. – ISBN 978-1-59593-795-7 (page 51)

[Ferdinand et al. 2001] Ferdinand, Christian ; Heckmann, Reinhold ; Langenbach, Marc ; Martin, Florian ; Schmidt, Michael ; Theiling, Henrik ; Thesing, Stephan ; Wilhelm, Reinhard: Reliable and Precise WCET Determination for a Real-Life Processor. In: *Proceedings of the First International Workshop on Embedded Software*. London, UK : Springer-Verlag, 2001 (EMSOFT '01), p. 469–485. – DOI http://dl.acm.org/citation.cfm?id=646787.703893. – ISBN 3-540-42673-6 (page 86)

[Foster 1995] Foster, Ian: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA : Addison-Wesley Longman, 1995. – ISBN 0201575949 (page 117, 137)

[Fraser 2004] Fraser, Keir: Practical lock-freedom / University of Cambridge, Computer Laboratory. February 2004 (UCAM-CL-TR-579). – Technical Report (page 51)

[Fraser and Harris 2007] Fraser, Keir ; Harris, Tim: Concurrent Programming Without Locks. In: *ACM Transactions on Computer Systems (TOCS)* 25 (2007), May, No. 2. – DOI http://doi.acm.org/10.1145/1233307.1233309. – ISSN 0734-2071 (page 47, 48, 50)

[Freescale P4080] FREESCALE SEMICONDUCTOR INC.: *P4080 Fact Sheet*. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080. – Last Retrieved: April 2013 (page 19)

[Gamma et al. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0-201-63361-2 (page 116)

[Gao and Hesselink 2007] GAO, Hui ; HESSELINK, Wim H.: A general lock-free algorithm using compare-and-swap. In: *Inf. Comput.* 205 (2007), February, No. 2, p. 225–241. – DOI http://dx.doi.org/10.1016/j.ic.2006.10.003. – ISSN 0890-5401 (page 16, 47, 49)

[Gebhard et al. 2011] GEBHARD, Gernot ; CULLMANN, Christoph ; HECKMANN, Reinhold: Software Structure and WCET Predictability. In: LUCAS, Philipp (Pub.) ; THIELE, Lothar (Pub.) ; TRIQUET, Benoit (Pub.) ; UNGERER, Theo (Pub.) ; WILHELM, Reinhard (Pub.): *Bringing Theory to Practice: Predictability and Performance in Embedded Systems* Vol. 18. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, p. 1–10. – DOI http://dx.doi.org/10.4230/OASIcs.PPES.2011.1. – ISBN 978-3-939897-28-6 (page 2, 54, 115)

[Gerdes et al. 2012a] GERDES, Mike ; KLUGE, Florian ; ROCHANGE, Christine ; UNGERER, Theo: The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs. In: *Proceedings of the eighteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA : IEEE Computer Society, August 2012 (RTCSA '12), p. 88–97. – DOI http://dx.doi.org/10.1109/RTCSA.2012.11. – ISSN 1533-2306 (page 23, 57, 89)

[Gerdes et al. 2012b] GERDES, Mike ; KLUGE, Florian ; UNGERER, Theo ; ROCHANGE, Christine ; SAINRAT, Pascal: Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In: *Proceedings of Design, Automation and Testing in Europe*. Leuven, Belgium : European Design and Automation Association, March 2012 (DATE '12), p. 671–676. – DOI http://dx.doi.org/10.1109/DATE.2012.6176555 (page 1, 2, 19, 23, 32, 57, 77, 79, 88)

[Gerdes et al. 2011] GERDES, Mike ; WOLF, Julian ; GULIASHVILI, Irakli ; UNGERER, Theo ; HOUSTON, Michael ; BERNAT, Guillem ; SCHNITZLER, Stefan ; REGLER, Hans: Large Drilling Machine Control Code - Parallelisation and WCET Speedup. In: *Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems*, DOI http://dx.doi.org/10.1109/SIES.2011.5953688, June 2011 (SIES '11), p. 91–94 (page 57, 88, 124)

[Girkar and Polychronopoulos 1994] Girkar, Milind ; Polychronopoulos, Constantine D.: The Hierarchical Task Graph as a Universal Intermediate Representation. In: *Int. J. Parallel Program.* 22 (1994), October, No. 5, p. 519–551. – DOI http://dx.doi.org/10.1007/BF02577777. – ISSN 0885-7458 (page 131)

[Gottlieb and Kruskal 1981] Gottlieb, Allan ; Kruskal, Clyde P.: Coordinating parallel processors: a partial unification. In: *SIGARCH Computer Architecture News* 9 (1981), October, No. 6, p. 16–24. – DOI http://doi.acm.org/10.1145/859515.859517. – ISSN 0163-5964 (page 14)

[Gottlieb et al. 1983] Gottlieb, Allan ; Lubachevsky, Boris D. ; Rudolph, Larry: Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5 (1983), April, No. 2, p. 164–189. – DOI http://doi.acm.org/10.1145/69624.357206. – ISSN 0164-0925 (page 14)

[Grand 2002] Grand, Mark: *Patterns in Java. A Catalog of Reusable Design Patterns Illustrated with UML.* 2nd Edition, Volume 1. Indianapolis, IN, USA : Wiley Publishing, Inc., September 2002. – ISBN 0471227293, 978-0471227298 (page 116)

[Graunke and Thakkar 1990] Graunke, Gary ; Thakkar, Shreekant: Synchronization Algorithms for Shared-Memory Multiprocessors. In: *IEEE Computer* 23 (1990), June, No. 6, p. 60–69. – DOI http://dx.doi.org/10.1109/2.55501. – ISSN 0018-9162 (page 45)

[Greenwald and Cheriton 1996] Greenwald, Michael ; Cheriton, David: The Synergy Between Non-Blocking Synchronization and Operating System Structure. In: *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation.* New York, NY, USA : ACM, 1996 (OSDI '96), p. 123–136. – DOI http://doi.acm.org/10.1145/238721.238767. – ISBN 1-880446-82-0 (page 47)

[Gustavsson et al. 2010] Gustavsson, Andreas ; Ermedahl, Andreas ; Lisper, Björn ; Pettersson, Paul: Towards WCET Analysis of Multicore Architectures using UPPAAL. In: Lisper, Björn (Pub.): *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)* Vol. 15. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, p. 101–112. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2010.101. – The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. – ISBN 978-3-939897-21-7 (page 1, 87)

[Gustavsson et al. 2012] Gustavsson, Andreas ; Gustafsson, Jan ; Lisper, Björn: Toward Static Timing Analysis of Parallel Software. In: Vardanega, Tullio (Pub.): *Proceedings of the 12th International Workshop on Worst-Case Execution-Time (WCET) Analysis* Vol. 23. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, July 2012, p. 38–47. – ISBN 978-3-939897-41-5 (page 2, 87)

[Ha-Hoai and Tsigas 2003] Ha-Hoai, Phuong ; Tsigas, Philippas: Fast, Reactive and Lock-free Multi-Word Compare-and-Swap Algorithms. 2003 (2003-06). – Technical Report (page 15, 47, 50)

[Hansson et al. 2009] Hansson, Andreas ; Goossens, Kees ; Bekooij, Marco ; Huisken, Jos: CoMPSoC: A template for composable and predictable multi-processor system on chips. In: *ACM Trans. Des. Autom. Electron. Syst.* 14 (2009), January, No. 1, p. 2:1–2:24. – DOI http://doi.acm.org/10.1145/1455229.1455231. – ISSN 1084-4309 (page 2, 86)

[Hardy et al. 2009] Hardy, Damien ; Piquet, Thomas ; Puaut, Isabelle: Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In: *30th IEEE Real-Time Systems Symposium.* Washington, DC, USA : IEEE Computer Society, December 2009 (RTSS '09), p. 68–77. – DOI http://dx.doi.org/10.1109/RTSS.2009.34. – ISBN 978-0-7695-3875-4 (page 2, 86)

[Harris et al. 2010] Harris, Tim ; Larus, James R. ; Rajwar, Ravi: *Transactional Memory.* 2nd Edition. San Rafael, CA, USA : Morgan and Claypool Publishers, 2010 (Synthesis Lectures on Computer Architecture). – ISBN 1608452352, 9781608452354 (page 50)

[Harris et al. 2002] Harris, Tim L. ; Fraser, Keir ; Pratt, Ian A.: A Practical Multi-Word Compare-and-Swap Operation. In: *Proceedings of the 16th International Symposium on Distributed Computing.* London, UK : Springer-Verlag, 2002 (DISC '02), p. 265–279. – DOI http://dl.acm.org/citation.cfm?id=645959.676137. – ISBN 3-540-00073-9 (page 15, 50)

[Harrison et al. 2000] Harrison, Neil ; Foote, Brian ; Rohnert, Hans: *Pattern Languages of Program Design.* Vol. 4. Boston, MA, USA : Addison Wesley, 2000. – ISBN 0201433044 (page 116)

[Heckmann et al. 2003] Heckmann, Reinhold ; Langenbach, Marc ; Thesing, Stephan ; Wilhelm, Reinhard: The influence of processor architecture on the design and the results of WCET tools. In: *Proceedings of the IEEE* 91 (2003), July, No. 7, p. 1038–1054. – DOI http://dx.doi.org/10.1109/JPROC.2003.814618. – ISSN 0018-9219 (page 86)

[Hennessy and Patterson 2003] Hennessy, John L. ; Patterson, David A.: *Computer Architecture - A Quantitative Approach.* 3rd Edition. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2003. – ISBN 978-1-55860-596-1 (page 8, 11, 13, 14, 27, 40, 41, 42, 43, 47)

[Hensgen et al. 1988] Hensgen, Debra ; Finkel, Raphael ; Manber, Udi: Two Algorithms for Barrier Synchronization. In: *International Journal on Parallel Programming* 17 (1988), February, No. 1, p. 1–17. – DOI http://dx.doi.org/10.1007/BF01379320. – ISSN 0885-7458 (page 42, 46)

[Herlihy 1988] Herlihy, Maurice P.: Impossibility and Universality Results for Wait-Free Synchronization. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of distributed computing.* New York, NY, USA : ACM, 1988 (PODC '88), p. 276–290. – DOI http://doi.acm.org/10.1145/62546.62593. – ISBN 0-89791-277-2 (page 47, 48)

[Herlihy 1991a] Herlihy, Maurice P.: Wait-Free Synchronization. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), January, No. 1, p. 124–149. – DOI http://doi.acm.org/10.1145/114005.102808. – ISSN 0164-0925 (page 47, 48)

[Herlihy 1991b] Herlihy, Maurice P.: Wait-Free Synchronization. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1991), January, No. 1, p. 124–149. – DOI http://doi.acm.org/10.1145/114005.102808. – ISSN 0164-0925 (page 49)

[Herlihy 1993] Herlihy, Maurice P.: A Methodology for Implementing Highly Concurrent Data Objects. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15 (1993), November, No. 5, p. 745–770. – DOI http://doi.acm.org/10.1145/161468.161469. – ISSN 0164-0925 (page 47, 48, 49)

[Herlihy et al. 2003] Herlihy, Maurice P. ; Luchangco, Victor ; Moir, Mark: Obstruction-Free Synchronization: Double-Ended Queues as an Example. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems.* Washington, DC, USA : IEEE Computer Society, 2003 (ICDCS '03), p. 522–. – DOI http://dl.acm.org/citation.cfm?id=850929.851942. – ISBN 0-7695-1920-2 (page 47, 48)

[Herlihy and Moss 1993] Herlihy, Maurice P. ; Moss, J. Eliot B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture.* New York, NY, USA : ACM, 1993 (ISCA '93), p. 289–300. – DOI http://doi.acm.org/10.1145/165123.165164. – ISBN 0-8186-3810-9 (page 15, 50)

[Herlihy and Wing 1990] Herlihy, Maurice P. ; Wing, Jeannette M.: Linearizability: A Correctness Condition for Concurrent Objects. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (1990), July, No. 3, p. 463–492. – DOI http://doi.acm.org/10.1145/78969.78972. – ISSN 0164-0925 (page 8)

[Holzmann 2006] Holzmann, Gerard J.: The Power of 10: Rules for Developing Safety-Critical Code. In: *IEEE Computer* 39 (2006), June, No. 6, p. 95–97. – DOI http://dx.doi.org/10.1109/MC.2006.212. – ISSN 0018-9162 (page 117)

[IBM z/Architecture 2005] IBM: *z/Architecture. Principles of Operation.* https://www-304.ibm.com/support/docview.wss?uid=isg2b9de5f05a9d57819852571c500428f9a. September 2005. – 5th edition, Last Retrieved: April 2013 (page 15)

[Infineon AURIX 2013] Infineon Technologies: *Infineon AURIX (AUtomotive Realtime Integrated NeXt Generation Architecture).* http://www.infineon.com/cms/en/product/channel.html?channel=db3a30433727a44301372b2eefbb48d9. 2013. – Last Retrieved: April 2013 (page 19, 137)

[Jahr et al. 2013a] Jahr, Ralf ; Gerdes, Mike ; Ungerer, Theo: A Pattern-supported Parallelization Approach. In: *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores.* New York, NY, USA : ACM, February 2013 (PMAM '13), p. 53–62. – DOI http://doi.acm.org/10.1145/2442992.2442998. – ISBN 978-1-4503-1908-9 (page 118, 131, 132, 138)

[Jahr et al. 2013b] Jahr, Ralf ; Gerdes, Mike ; Ungerer, Theo: An Approach for Parallelisation with Parallel Design Patterns. In: *Journal on Parallel Computing (ParCo), Special Issue* (2013). – Submitted and under review (page 118, 131, 138)

[Jensen et al. 1987] Jensen, Eric H. ; Hagensen, Gary W. ; Broughton, Jeffrey M.: A New Approach to Exclusive Data Access in Shared Memory Multiprocessors / Lawrence Livermore National Laboratory. November 1987 (UCRL-97663). – Technical Report (page 15)

[Kelter et al. 2011] Kelter, Timon ; Falk, Heiko ; Marwedel, Peter ; Chattopadhyay, Sudipta ; Roychoudhury, Abhik: Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems.* Washington, DC, USA : IEEE Computer Society, July 2011 (ECRTS '11), p. 3–12. – DOI http://dx.doi.org/10.1109/ECRTS.2011.9. – ISSN 1068-3070 (page 2, 19, 86)

[Keutzer et al. 2010] Keutzer, Kurt ; Massingill, Berna L. ; Mattson, Timothy G. ; Sanders, Beverly A.: A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns.* New York, NY, USA : ACM, 2010 (ParaPLoP '10), p. 9:1–9:8. – DOI http://doi.acm.org/10.1145/1953611.1953620. – ISBN 978-1-4503-0127-5 (page 119)

[Kirner and Puschner 2010] Kirner, Raimund ; Puschner, Peter: Time-Predictable Computing. In: *Proceedings of the 8th IFIP WG 10.2 International Conference on Software technologies for Embedded and Ubiquitous Systems.* Berlin / Heidelberg, Germany : Springer-Verlag, 2010 (SEUS '10), p. 23–34. – DOI http://dl.acm.org/citation.cfm?id=1927882.1927890. – ISBN 3-642-16255-X, 978-3-642-16255-8 (page 86, 135, 138)

[Knuth 1966] Knuth, Donald E.: Additional Comments on a Problem in Concurrent Programming Control. In: *Communications of the ACM* 9 (1966), May, No. 5, p. 321–322. – DOI http://doi.acm.org/10.1145/355592.365595. – ISSN 0001-0782 (page 10)

[Kopetz and Nossal 1997] Kopetz, Hermann ; Nossal, Roman: Temporal Firewalls in Large Distributed Real-Time Systems. In: *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*. Los Alamitos, CA, USA : IEEE Computer Society, 1997 (FTDCS '97), p. 310–. – DOI http://dx.doi.org/10.1109/FTDCS.1997.644743. – ISSN 1071-0485 (page 138)

[Kowalik 1985] Kowalik, Janusz S.: *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*. Cambridge, MA, USA : The MIT Press, 1985. – ISBN 0-262-11101-2 (page 17)

[Kruskal et al. 1988] Kruskal, Clyde P. ; Rudolph, Larry ; Snir, Marc: Efficient Synchronization of Multiprocessors with Shared Memory. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10 (1988), October, p. 579–601. – DOI http://doi.acm.org/10.1145/48022.48024. – ISSN 0164-0925 (page 8, 14, 23, 27)

[Lakis and Schoeberl 2013] Lakis, Edgar ; Schoeberl, Martin: An SDRAM Controller for Real-Time Systems. In: *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013 (SEUS '13) (page 25)

[Lamport 1974] Lamport, Leslie: A New Solution of Dijkstra's Concurrent Programming Problem. In: G.Bell, S.H. F. (Pub.): *Communications of the ACM* Vol. 17. New York, NY, USA : ACM, August 1974, p. 453–455. – DOI http://doi.acm.org/10.1145/361082.361093 (page 13)

[Lamport 1979] Lamport, Leslie: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. In: *IEEE Transactions on Computers* C-28 (1979), September, No. 9, p. 690–691. – DOI http://dx.doi.org/10.1109/TC.1979.1675439. – ISSN 0018-9340 (page 8, 23)

[Larus and Rajwar 2007] Larus, James R. ; Rajwar, Ravi ; Hill, Mark D. (Pub.): *Transactual Memory*. First Edition. San Rafael, CA, USA : Morgan & Claypool, 2007. – ISBN 1598291246 (page 50)

[Lea 2003] Lea, Doug: *Concurrent Programming in JAVA*. 2nd Edition. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0-201-31009-0 (page 116)

[Lehoczky and Ramos-Thuel 1992] Lehoczky, John P. ; Ramos-Thuel, Sandra: An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In: *Proceedings of Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, December 1992 (RTSS '92), p. 110–123. – DOI http://dx.doi.org/REAL.1992.242671 (page 21)

[Li et al. 2007] Li, Xianfeng ; Liang, Yun ; Mitra, Tulika ; Roychoudhury, Abhik: Chronos: A Timing Analyzer for Embedded Software. In: *Sci. Comput. Program.* 69 (2007), December, No. 1-3, p. 56–67. – DOI http://dx.doi.org/10.1016/j.scico.2007.01.014. – ISSN 0167-6423 (page 86)

[Li et al. 2009] Li, Yan ; Suhendra, V. ; Liang, Yun ; Mitra, T. ; Roychoudhury, A.: Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In: *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2009 (RTSS '09), p. 57–67. – DOI http://dx.doi.org/10.1109/RTSS.2009.32. – ISBN 978-0-7695-3875-4 (page 87)

[Lickly et al. 2008] Lickly, Ben ; Liu, Isaac ; Kim, Sungjun ; Patel, Hiren D. ; Edwards, Stephen A. ; Lee, Edward A.: Predictable Programming on a Precision Timed Architecture. In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY, USA : ACM, 2008 (CASES '08), p. 137–146. – DOI http://doi.acm.org/10.1145/1450095.1450117. – ISBN 978-1-60558-469-0 (page 2, 87)

[Lisper 2012] Lisper, Björn: Towards Parallel Programming Models for Predictability. In: Vardanega, Tullio (Pub.): *Proceedings of the 12th International Workshop on Worst-Case Execution-Time (WCET) Analysis* Vol. 23. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, p. 48–58. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2012.48. – ISBN 978-3-939897-41-5 (page 87)

[Liu and Layland 1973] Liu, Chang L. ; Layland, James W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In: *Journal of the ACM (JACM)* 20 (1973), January, No. 1, p. 46–61. – DOI http://doi.acm.org/10.1145/321738.321743. – ISSN 0004-5411 (page 21)

[Liu 2012] Liu, Isaac: *Precision Timed Machines*, EECS Department, University of California, Berkeley, phd thesis, May 2012 (page 19)

[Liu et al. 2010] Liu, Isaac ; Reineke, Jan ; Lee, Edward A.: A PRET architecture supporting concurrent programs with composable timing properties. In: *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, DOI http://dx.doi.org/10.1109/ACSSC.2010.5757922, November 2010 (ASILOMAR '10), p. 2111–2115. – ISSN 1058-6393 (page 2, 19)

[Liu and Gaudiot 2007] Liu, Shaoshan ; Gaudiot, Jean-Luc: Synchronization Mechanisms on Modern Multi-core Architectures. Berlin / Heidelberg, Germany : Springer-Verlag, 2007 (ACSAC'07), p. 290–303. – DOI http://dx.doi.org/10.1007/978-3-540-74309-5_28. – ISBN 3-540-74308-1, 978-3-540-74308-8 (page 101)

[Lubachevsky 1984] Lubachevsky, Boris D.: An Approach to Automating the Verification of Compact Parallel Coordination Programs. In: *Acta Informatica* 21 (1984), p. 125–169. – DOI http://dx.doi.org/10.1007/BF00289237. – ISSN 0001-5903 (page 44)

[LynxSecure] LynuxWorks: *LynxSecure Embedded Hypervisor*. http://www.lynuxworks.com/virtualization/hypervisor.php. – Last Retrieved: April 2013 (page 2)

[Maldonado et al. 2011] Maldonado, Walther ; Marlier, Patrick ; Felber, Pascal ; Lawall, Julia ; Muller, Giller ; Rivière, Etienne: Deadline-Aware Scheduling for Software Transactional Memory. In: *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems Networks*. Washington, DC, USA : IEEE Computer Society, June 2011 (DSN '11), p. 257–268. – DOI http://dx.doi.org/10.1109/DSN.2011.5958224. – ISSN 1530-0889 (page 50, 51)

[Manolescu et al. 2006] Manolescu, Dragos ; Voelter, Markus ; Noble, James: *Pattern Languages of Program Design*. Vol. 5. Upper Saddle River, NJ, USA : Addison-Wesley, 2006. – ISBN 0321321944 (page 116)

[Manson et al. 2005] Manson, Jeremy ; Baker, Jason ; Cunei, Antonio ; Jagannathan, Suresh ; Prochazka, Marek ; Xin, Bin ; Vitek, Jan: Preemptible Atomic Regions for Real-Time Java. In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2005 (RTSS '05), p. 62–71. – DOI http://dx.doi.org/10.1109/RTSS.2005.34. – ISBN 0-7695-2490-7 (page 50)

[Marejka 1994] Marejka, Richard: A Barrier for Threads. In: *SunOpsis - The Solaris 2.0 Migration Support Centre Newsletter* 4 (1994), November, No. 1 (page 42, 75, 148)

[Markatos et al. 1991] Markatos, Evangelos P. ; Crovella, Mark ; Das, Prakash ; Dubnicki, Cezary ; LeBlanc, Thomas J.: The Effects of Multiprogramming on Barrier Synchronization. In: *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*. Los Alamitos, CA, USA : IEEE Computer Society Press, December 1991, p. 662–669. – DOI http://dx.doi.org/10.1109/SPDP.1991.218199 (page 46)

[Martin et al. 1997] Martin, Robert C. (Pub.) ; Riehle, Dirk (Pub.) ; Buschmann, Frank (Pub.): *Pattern Languages of Program Design*. Vol. 3. Boston, MA, USA : Addison-Wesley, 1997. – ISBN 0-201-31011-2 (page 116)

[Massalin and Pu 1992] Massalin, Henry ; Pu, Calton: A Lock-Free Multiprocessor OS Kernel. In: *SIGOPS Oper. Syst. Rev.* 26 (1992), April, No. 2, p. 8–. – DOI http://dl.acm.org/citation.cfm?id=142111.993246. – ISSN 0163-5980 (page 48)

[Massingill et al. 1999] Massingill, Berna L. ; Mattson, Timothy G. ; Sanders, Beverly A.: Patterns for Parallel Application Programs. In: *Proceedings of the 6th Pattern Languages of Programs Workshop*, August 1999 (PLoP '99) (page 117)

[Massingill et al. 2001a] Massingill, Berna L. ; Mattson, Timothy G. ; Sanders, Beverly A.: More Patterns for Parallel Application Programs. In: *Proceedings of the 8th Pattern Languages of Programs Workshop*, September 2001 (PLoP '01) (page 117)

[Massingill et al. 2001b] Massingill, Berna L. ; Mattson, Timothy G. ; Sanders, Beverly A.: Parallel programming with a pattern language. In: *International Journal* *on Software Tools for Technology Transfer (STTT)* 3 (2001), No. 2, p. 217–234. – DOI http://dx.doi.org/10.1007/s100090100045 (page 117)

[Mattson et al. 2010] Mattson, Timothy G. ; Riepen, Michael ; Lehnig, Thomas ; Brett, Paul ; Haas, Werner ; Kennedy, Patrick ; Howard, Jason ; Vangal, Sriram ; Borkar, Nitin ; Ruhl, Greg ; Dighe, Saurabh: The 48-core SCC Processor: the Programmer's View. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* Washington, DC, USA : IEEE Computer Society, 2010 (SC '10), p. 1–11. – DOI http://dx.doi.org/10.1109/SC.2010.53. – ISBN 978-1-4244-7559-9 (page 17, 22)

[Mattson et al. 2004] Mattson, Timothy G. ; Sanders, Beverly A. ; Massingill, Berna L.: *Patterns for Parallel Programming.* 1st Edition. Boston, MA, USA : Addison-Wesley Professional, 2004. – ISBN 0321228111 (page 117, 118, 119, 137)

[Meawad et al. 2011] Meawad, Fadi ; Schoeberl, Martin ; Iyer, Karthik ; Vitek, Jan: Real-Time Wait-Free Queues Using Micro-Transactions. In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* New York, NY, USA : ACM, 2011 (JTRES '11), p. 1–10. – DOI http://doi.acm.org/10.1145/2043910.2043912. – ISBN 978-1-4503-0731-4 (page 50)

[Mellor-Crummey and Scott 1991a] Mellor-Crummey, John M. ; Scott, Michael L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In: *ACM Transactions on Computer Systems* 9 (1991), February, No. 1, p. 21–65. – DOI http://doi.acm.org/10.1145/103727.103729. – ISSN 0734-2071 (page 8, 11, 13, 14, 27, 36, 45)

[Mellor-Crummey and Scott 1991b] Mellor-Crummey, John M. ; Scott, Michael L.: Synchronization Without Contention. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91).* New York, NY, USA : ACM, 1991 (ASPLOS-IV), p. 269–278. – DOI http://doi.acm.org/10.1145/106972.106999. – ISBN 0-89791-380-9 (page 46)

[Metzlaff 2012] Metzlaff, Stefan: *Analysable Instruction Memories for Hard Real-Time Systems*, Computer Science Department, University of Augsburg, Germany, phd thesis, May 2012 (page 55)

[Metzlaff et al. 2011] Metzlaff, Stefan ; Guliashvili, Irakli ; Uhrig, Sascha ; Ungerer, Theo: A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware. In: *Proceedings of the 24th International Conference on Architecture of Computing Systems.* Berlin / Heidelberg, Germany : Springer-Verlag, 2011 (ARCS'11), p. 122–134. – DOI http://dl.acm.org/citation.cfm?id=1966221.1966236. – ISBN 978-3-642-19136-7 (page 20, 89)

[Michael 2004] Michael, Maged M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. In: *IEEE Transactions on Parallel Distributed Systems* 15 (2004), June, No. 6, p. 491–504. – DOI http://dx.doi.org/10.1109/TPDS.2004.8. – ISSN 1045-9219 (page 15, 16, 47, 49)

[Milewski 1990] MILEWSKI, Jaroslaw: Functional Data Structures as Updatable Objects. In: *IEEE Transactions on Software Engineering* 16 (1990), December, No. 12, p. 1427–1432. – DOI http://dx.doi.org/10.1109/32.62450. – ISSN 0098-5589 (page 18)

[MIPS32 ISA 2003] MIPS TECHNOLOGIES, INC.: *MIPS32 Instruction Set Architecture Revision 3.05*. http://www.mips.com/products/architectures/mips32/. June 2003. – Last Retrieved: April 2013 (page 16, 18, 100)

[Mische et al. 2010] MISCHE, Jörg ; GULIASHVILI, Irakli ; UHRIG, Sascha ; UNGERER, Theo: How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT. In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems*. Berlin / Heidelberg, Germany : Springer-Verlag, 2010 (ARCS'10), p. 2–14. – DOI http://dx.doi.org/10.1007/978-3-642-11950-7_2. – ISBN 3-642-11949-2, 978-3-642-11949-1 (page 17, 20, 47)

[Mittermayr and Blieberger 2012] MITTERMAYR, Robert ; BLIEBERGER, Johann: Timing Analysis of Concurrent Programs. In: VARDANEGA, Tullio (Pub.): *Proceedings of the 12th International Workshop on Worst-Case Execution-Time (WCET) Analysis* Vol. 23. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, p. 59–68. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2012.59. – ISBN 978-3-939897-41-5 (page 87)

[Mohan et al. 2011] MOHAN, Sibin ; CACCAMO, Marco ; SHA, Lui ; PELLIZZONI, Rodolfo ; ARUNDALE, Greg ; KEGLEY, Russell ; NIZ, Dionisio de: Using Multicore Architectures in Cyber-Physical Systems. In: *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*, March 2011 (page 86)

[Molesky et al. 1990] MOLESKY, Lory D. ; SHEN, Chia ; ZLOKAPA, Goran: Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems. In: *Real-Time Systems* 2 (1990), September, No. 3, p. 163–180. – DOI http://dx.doi.org/10.1007/BF00365325. – ISSN 0922-6443 (page 23, 44)

[Monchiero et al. 2005] MONCHIERO, Matteo ; PALERMO, Gianluca ; SILVANO, Cristina ; VILLA, Oreste: An Efficient Synchronization Technique for Multiprocessor Systems on-Chip. In: *Proceedings of the 2005 Workshop on MEmory performance: DEaling with Applications, systems and architecture*. Washington, DC, USA : IEEE Computer Society, 2005 (MEDEA '05), p. 33–40. – DOI http://dx.doi.org/10.1145/1147349.1147357 (page 100)

[Mukherjee 2008] MUKHERJEE, Suhubu: *Architecture Design for Soft Errors*. First Edition. Burlington, MA, USA : Morgan Kaufmann Publishers, 2008. – ISBN 978-0-12-369529-1 (page 137)

[Mutlu and Moscibroda 2007] Mutlu, Onur ; Moscibroda, Thomas: Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2007 (MICRO 40), p. 146–160. – DOI http://dx.doi.org/10.1109/MICRO.2007.40. – ISBN 0-7695-3047-8 (page 25)

[Nichols et al. 1996] Nichols, Bradford ; Buttlar, Dick ; Proulx Farrell, Jacqueline: *Pthreads Programming*. First Edition. Sebastopol, CA, USA : O'Reilly & Associates, Inc., September 1996. – ISBN 1-56592-115-1 (page 38)

[Nikhil 1991] Nikhil, Rishiyur S.: ID Language Reference Manual (Version 90.1) / MIT Lab. for Computer Science, Massachusetts Institute of Technology, Cambridge, USA. July 1991 (CSG Memo 284-2). – Technical Report (page 18)

[Nowotsch and Paulitsch 2012] Nowotsch, Jan ; Paulitsch, Michael: Leveraging Multi-core Computing Architectures in Avionics. In: *European Dependable Computing Conference*. Los Alamitos, CA, USA : IEEE Computer Society, 2012 (EDCC '12), p. 132–143. – DOI http://doi.ieeecomputersociety.org/10.1109/EDCC.2012.27. – ISBN 978-0-7695-4671-1 (page 2, 19)

[Olukotun 2006] Olukotun, Kunle: A Conversation with John Hennessy and David Patterson. In: *Queue - Computer Architecture* 4 (2006), December, No. 10. – ISSN 1542-7730 (page 117)

[Ozaktas et al. 2013] Ozaktas, Haluk ; Rochange, Christine ; Sainrat, Pascal: Automatic WCET Analysis of Real-Time Parallel Applications. In: Maiza, Claire (Pub.): *Proceedings of the 13th International Workshop on Worst-Case Execution Time (WCET) Analysis* Vol. 30. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, p. 11–20. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2013.11. – ISBN 978-3-939897-54-5 (page 54, 131)

[Paap and Silha 1993] Paap, George ; Silha, Ed: PowerPC^{TM}: A Performance Architecture. In: *Compcon Spring '93, Digest of Papers.*, DOI http://dx.doi.org/10.1109/CMPCON.1993.289645, February 1993, p. 104–108 (page 15)

[Paolieri et al. 2013] Paolieri, Marco ; Mische, Jörg ; Metzlaff, Stefan ; Gerdes, Mike ; Quiñones, Eduardo ; Uhrig, Sascha ; Ungerer, Theo ; Cazorla, Francisco J.: A Hard Real-Time Capable Multi-Core SMT Processor. In: *ACM Trans. Embed. Comput. Syst.* 12 (2013), April, No. 3, p. 79:1–79:26. – DOI http://doi.acm.org/http://dx.doi.org/10.1145/2442116.2442129. – ISSN 1539-9087 (page 16, 19, 20, 26, 33, 62)

[Paolieri et al. 2009a] Paolieri, Marco ; Quiñones, Eduardo ; Cazorla, Francisco J. ; Bernat, Guillem ; Valero, Mateo: Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2009

(ISCA '09), p. 57–68. – DOI http://doi.acm.org/10.1145/1555754.1555764. – ISBN 978-1-60558-526-0 (page 2, 19, 20, 26, 33, 44, 57, 62, 86)

[Paolieri et al. 2009b] PAOLIERI, Marco ; QUIÑONES, Eduardo ; CAZORLA, Francisco J. ; VALERO, Mateo: An Analyzable Memory Controller for Hard Real-Time CMPs. In: *Embedded Systems Letters, IEEE* 1 (2009), dec., No. 4, p. 86–90. – DOI http://dx.doi.org/10.1109/LES.2010.2041634. – ISSN 1943-0663 (page 2, 25)

[Peterson 1981] PETERSON, Gary L.: Myths About the Mutual Exclusion Problem. In: *Inf. Process. Lett.* 12 (1981), No. 3, p. 115–116. – DOI http://dx.doi.org/10.1016/0020-0190(81)90106-X (page 22)

[Petters et al. 2007] PETTERS, Stefan M. ; ZADARNOWSKI, Patryk ; HEISER, Gernot: Measurements or Static Analysis or Both? In: ROCHANGE, Christine (Pub.): *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Dagstuhl, Germany : Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2007.1188 (page 57)

[Pitter 2008] PITTER, Christof: Time-Predictable Memory Arbitration for a Java Chip-Multiprocessor. In: *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems*. New York, NY, USA : ACM, 2008 (JTRES '08), p. 115–122. – DOI http://doi.acm.org/10.1145/1434790.1434808. – ISBN 978-1-60558-337-2 (page 2)

[Pitter and Schoeberl 2010] PITTER, Christof ; SCHOEBERL, Martin: A Real-Time Java Chip-Multiprocessor. In: *ACM Trans. Embed. Comput. Syst.* 10 (2010), August, No. 1, p. 9:1–9:34. – DOI http://doi.acm.org/10.1145/1814539.1814548. – ISSN 1539-9087 (page 2)

[POSIX 2008] POSIX 2008: *IEEE Std 1003.1, 2008 Edition. The Open Group Base Specifications Issue 7*. 2008 (page 10, 11, 13, 22, 32, 35, 38, 98)

[PowerPC ISA 2010] POWER.ORG: *Power Instruction Set Architecture v2.06 Revision B*. https://www.power.org/resources/reading/. July 2010. – Last Retrieved: April 2013 (page 16, 18, 31, 100, 133)

[Puschner and Burns 2000] PUSCHNER, Peter ; BURNS, Alan: A Review of Worst-Case Execution-Time Analysis. In: *Journal of Real-Time Systems* 18 (2000), May, No. 2/3, p. 115–128 (page 55)

[Puschner and Schoeberl 2008] PUSCHNER, Peter ; SCHOEBERL, Martin: On Composable System Timing, Task Timing, and WCET Analysis. In: KIRNER, Raimund (Pub.): *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Dagstuhl, Germany : Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2008.1662. – also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3. – ISBN 978-3-939897-10-1 (page 2, 55, 57, 86)

[Rajkumar 1990] RAJKUMAR, Ragunathan: Real-Time Synchronization Protocols for Shared Memory Multiprocessors. In: *Proceedings of the 10th International Conference on Distributed Computing Systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, June 1990, p. 116–123. – DOI http://dx.doi.org/10.1109/ICDCS.1990.89257 (page 47)

[Rajkumar et al. 1988] RAJKUMAR, Ragunathan ; SHA, Lui ; LEHOCZKY, John P.: Real-Time Synchronization Protocols for Multiprocessors. In: *Proceedings of the Real-Time Systems Symposium*. Los Alamitos, CA, USA : IEEE Computer Society Press, December 1988, p. 259–269. – DOI http://dx.doi.org/10.1109/REAL.1988.51121 (page 47)

[Ramakrishnan and Scherson 1999] RAMAKRISHNAN, Vara ; SCHERSON, Isaac D.: Efficient Techniques for Nested and Disjoint Barrier Synchronization. In: *Journal of Parallel and Distributed Computing - Special issue on compilation and architectural support for parallel applications* 58 (1999), August, No. 2, p. 333–356. – DOI http://dx.doi.org/10.1006/jpdc.1999.1556. – ISSN 0743-7315 (page 46)

[Rapita Systems Ltd. 2011] RAPITA SYSTEMS LTD.: *RapiTime White Paper*. http://www.rapitasystems.com/downloads/white-papers/rapitime-explained. May 2011. – Last Retrieved: April 2013 (page 7, 57, 88)

[Reineke et al. 2011] REINEKE, Jan ; LIU, Isaac ; PATEL, Hiren D. ; KIM, Sungjun ; LEE, Edward A.: PRET DRAM controller: bank privatization for predictability and temporal isolation. In: *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA : ACM, 2011 (CODES+ISSS '11), p. 99–108. – DOI http://doi.acm.org/10.1145/2039370.2039388. – ISBN 978-1-4503-0715-4 (page 2, 25)

[Reineke and Sen 2009] REINEKE, Jan ; SEN, Rathijit: Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In: HOLSTI, Niklas (Pub.): *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 2009. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2009.2289. – also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. – ISBN 978-3-939897-14-9 (page 55, 88)

[Rochange 2011] ROCHANGE, Christine: An Overview of Approaches Towards the Timing Analysability of Parallel Architecture. In: LUCAS, Philipp (Pub.) ; THIELE, Lothar (Pub.) ; TRIQUET, Benoit (Pub.) ; UNGERER, Theo (Pub.) ; WILHELM, Reinhard (Pub.): *Bringing Theory to Practice: Predictability and Performance in Embedded Systems* Vol. 18. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, p. 32–41. – DOI http://dx.doi.org/10.4230/OASIcs.PPES.2011.32. – ISBN 978-3-939897-28-6 (page 1, 2, 55)

[Rochange et al. 2010] ROCHANGE, Christine ; BONENFANT, Armelle ; SAINRAT, Pascal ; GERDES, Mike ; WOLF, Julian ; UNGERER, Theo ; PETROV, Zlatko ;

Mikulu, Frantisek: WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In: Lisper, Björn (Pub.): *Proceedings of the 10th International Workshop on Worst-Case Execution Time (WCET) Analysis* Vol. 15. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, July 2010, p. 90–100. – DOI http://dx.doi.org/10.4230/OASIcs.WCET.2010.90. – The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. – ISBN 978-3-939897-21-7 (page 2, 57, 77, 88)

[Rosen et al. 2007] Rosen, Jakob ; Andrei, Alexandru ; Eles, Petru ; Peng, Zebo: Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: *28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, December 2007 (RTSS '07), p. 49–60. – DOI http://dx.doi.org/10.1109/RTSS.2007.24. – ISSN 1052-8725 (page 86)

[Rudolph and Segall 1984] Rudolph, Larry ; Segall, Zary: Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 1984 (ISCA '84), p. 340–347. – DOI http://doi.acm.org/10.1145/800015.808203. – ISBN 0-8186-0538-3 (page 14)

[Saha et al. 2006] Saha, Bratin ; Adl-Tabatabai, Ali-Reza ; Hudson, Richard L. ; Minh, Chi C. ; Hertzberg, Benjamin: McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : ACM, 2006 (PPoPP '06), p. 187–197. – DOI http://doi.acm.org/10.1145/1122971.1123001. – ISBN 1-59593-189-9 (page 50)

[Sandström et al. 1998] Sandström, Kristian ; Norström, Christer ; Fohler, Gerhard: Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In: *Proceedings of the Fifth International Conference on Real-Time Computing Systems and Applications*. Washington, DC, USA : IEEE Computer Society, October 1998 (RTCSA '98), p. 158–165. – DOI http://dl.acm.org/citation.cfm?id=600376.828679. – ISBN 0-8186-9209-X (page 21)

[Sarni et al. 2009] Sarni, Toufik ; Queudet, Audrey ; Valduriez, Patrick: Real-Time Support for Software Transactional Memory. In: *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA : IEEE Computer Society, August 2009 (RTCSA '09), p. 477–485. – DOI http://dx.doi.org/10.1109/RTCSA.2009.57. – ISBN 978-0-7695-3787-0 (page 50, 51)

[Sartori and Kumar 2010] Sartori, John ; Kumar, Rakesh: Low-Overhead, High-Speed Multi-core Barrier Synchronization. In: *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers* Vol. 5952.

Berlin / Heidelberg, Germany : Springer-Verlag, 2010, p. 18–34. – DOI `http://dx.doi.org/10.1007/978-3-642-11515-8`. – ISBN 978-3-642-11514-1 (page 45, 46)

[Schoeberl 2012] Schoeberl, Martin: Is Time Predictability Quantifiable? In: *International Conference on Embedded Computer Systems.* Los Alamitos, CA, USA : IEEE Computer Society Press, 2012 (SAMOS '12), p. 333–338. – DOI `http://dx.doi.org/10.1109/SAMOS.2012.6404196`. – ISBN 978-1-4673-2295-9 (page 86)

[Schoeberl et al. 2010] Schoeberl, Martin ; Brandner, Florian ; Vitek, Jan: RTTM: Real-Time Transactional Memory. In: *Proceedings of the 2010 ACM Symposium on Applied Computing.* New York, NY, USA : ACM, 2010 (SAC '10), p. 326–333. – DOI `http://doi.acm.org/10.1145/1774088.1774158`. – ISBN 978-1-60558-639-7 (page 50)

[Schoeberl and Hilber 2010] Schoeberl, Martin ; Hilber, Peter: Design and Implementation of Real-Time Transactional Memory. In: *Proceedings of International Conference on Field Programmable Logic and Applications.* Washington, DC, USA : IEEE Computer Society, September 2010 (FPL '10), p. 279–284. – DOI `http://dx.doi.org/10.1109/FPL.2010.64`. – ISSN 1946-1488 (page 50)

[Schoeberl and Puschner 2009] Schoeberl, Martin ; Puschner, Peter: Is Chip-Multiprocessing the End of Real-Time Scheduling? In: Holsti, Niklas (Pub.): *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis.* Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 2009. – DOI `http://dx.doi.org/10.4230/OASIcs.WCET.2009.2288`. – also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. – ISBN 978-3-939897-14-9 (page 45)

[Schranzhofer et al. 2010] Schranzhofer, Andreas ; Chen, Jian-Jia ; Thiele, Lothar: Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In: *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium.* Washington, DC, USA : IEEE Computer Society, 2010 (RTAS '10), p. 215–224. – DOI `http://dx.doi.org/10.1109/RTAS.2010.24`. – ISBN 978-0-7695-4001-6 (page 86)

[Schwartz 1980] Schwartz, Jacob T.: Ultracomputers. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2 (1980), October, No. 4, p. 484–521. – DOI `http://doi.acm.org/10.1145/357114.357116`. – ISSN 0164-0925 (page 44)

[Scott and Scherer 2001] Scott, Michael L. ; Scherer, William N.: Scalable Queue-Based Spin Locks with Timeout. In: *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming.* New York, NY, USA : ACM, 2001 (PPoPP '01), p. 44–52. – DOI `http://doi.acm.org/10.1145/379539.379566`. – ISBN 1-58113-346-4 (page 13, 45)

[Sha et al. 1990] Sha, Lui ; Rajkumar, Ragunathan ; Lehoczky, John P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In: *IEEE Transactions on Computers* 39 (1990), September, No. 9, p. 1175–1185. – DOI http://dx.doi.org/10.1109/12.57058. – ISSN 0018-9340 (page 47, 50)

[Sha et al. 1991] Sha, Lui ; Rajkumar, Ragunathan ; Son, S.H. ; Chang, C.-H.: A Real-Time Locking Protocol. In: *IEEE Transactions on Computers* 40 (1991), jul, No. 7, p. 793–800. – DOI http://dx.doi.org/10.1109/12.83617. – ISSN 0018-9340 (page 47, 50)

[Shang and Hwang 1995] Shang, Shisheng ; Hwang, Kai: Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters. In: *IEEE Transactions on Parallel Distributed Systems* 6 (1995), June, No. 6, p. 591–605. – DOI http://dx.doi.org/10.1109/71.388040. – ISSN 1045-9219 (page 46)

[Sites 1993] Sites, Richard L.: Alpha AXP architecture. In: *Commun. ACM* 36 (1993), February, p. 33–44. – DOI http://doi.acm.org/10.1145/151220.151226. – ISSN 0001-0782 (page 100)

[Skillicorn et al. 1997] Skillicorn, David B. ; Hill, Jonathan M. D. ; McColl, William F.: Questions and Answers about BSP. In: *Journal on Scientific Programming* 6 (1997), No. 3, p. 249–274. – DOI http://iospress.metapress.com/content/g31p42687n715641/ (page 87)

[Smith 1981] Smith, Burton J.: Architecture and Application of the HEP Multiprocessor Computer System. In: *Proceedings of SPIE, Real Time Signal Prcoessing IV* Vol. 298, August 1981, p. 241–248 (page 17, 18)

[SPARCv9 ISA 1994] SPARC International, Inc.: *SPARC Instruction Set Architecture v9*. Englewood Cliffs, NJ, USA : PTR Prentice Hall, 1994. – http://www.sparc.org/specificationsDownload.html. – Last Retrieved: April 2013. – ISBN 0-13-825001-4 (page 16, 18)

[Spuri and Buttazzo 1994] Spuri, Marco ; Buttazzo, Giorgio C.: Efficient Aperiodic Service under Earliest Deadline Scheduling. In: *Proceedings of Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, December 1994 (RTSS '94), p. 2–11. – DOI http://dx.doi.org/10.1109/REAL.1994.342735 (page 21)

[Stankovic 1988] Stankovic, John A.: Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems. In: *IEEE Transactions on Computer* 21 (1988), October, No. 10, p. 10–19. – DOI http://dx.doi.org/10.1109/2.7053. – ISSN 0018-9162 (page 5)

[Stankovic and Ramamritham 1990] Stankovic, John A. ; Ramamritham, Krithi: What is Predictability for Real-Time Systems? In: *Real-Time Systems* 2 (1990), No. 4, p. 247–254. – DOI http://dx.doi.org/10.1007/BF01995673. – ISSN 0922-6443 (page 1, 5)

[Staschulat et al. 2007] Staschulat, Jan ; Schliecker, Simon ; Ivers, Matthias ; Ernst, Rolf: Analysis of Memory Latencies in Multi-Processor Systems. In: Wilhelm, Reinhard (Pub.): *Proceedings of the 5th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Dagstuhl, Germany : Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2007, p. 33–36. – DOI `http://dx.doi.org/10.4230/OASIcs.WCET.2005.813` (page 2, 86)

[SuperH-2A ISA 2010] Renesas Electronics Corporation: *SH7265 Group Hardware Manual Revision 2.0*. `http://www.renesas.com/products/mpumcu/superh/Documentation.jsp`. February 2010. – Last Retrieved: April 2013 (page 16, 17, 19)

[SystemC 2007] Open SystemC Initiative (OSCI): *SystemC v2.2*. `http://www.systemc.org/downloads/standards/`. 2007. – Last Retrieved: April 2013 (page 20)

[Tanenbaum 2001] Tanenbaum, Andrew S.: *Modern Operating Systems*. International Edition of 2nd Edition. Upper Saddle River, NJ, USA : Prentice Hall International, 2001. – ISBN 0-13-092641-8 (page 10)

[Taubenfeld 2006] Taubenfeld, Gadi: *Synchronization Algorithms and Concurrent Programming*. Harlow, New York, NY, USA : Pearson / Prentice Hall, 2006. – ISBN 0131972596 (page 9)

[Taubenfeld 2008] Taubenfeld, Gadi: Concurrent Programming, Mutual Exclusion. In: *Encyclopedia of Algorithms* (2008). ISBN 978-0-387-30162-4 (page 7, 8)

[Thiele and Wilhelm 2004] Thiele, Lothar ; Wilhelm, Reinhard: Design for Time-Predictability. In: Thiele, Lothar (Pub.) ; Wilhelm, Reinhard (Pub.): *Perspectives Workshop: Design of Systems with Predictable Behaviour*, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004 (Dagstuhl Seminar Proceedings 03471), p. 157–177. – DOI `http://drops.dagstuhl.de/opus/volltexte/2004/2`. – ISSN 1862-4405 (page 2, 53, 86)

[Torvalds and Diamond 2002] Torvalds, Linus ; Diamond, David: *Just for Fun: The Story of an Accidental Revolutionary*. New York, NY, USA : HarperCollins, 2002. – ISBN 9780066620732 (page 22)

[TriCore 1.3 2002] Infineon Technologies AG: . `http://www.infineon.com/dgdl/TC1_3_ArchOverview_1.pdf?folderId=db3a304312bae05f0112be5d1a8100ec&fileId=db3a304312bae05f0112be86204c0111`. May 2002. – Last Retrieved: April 2013 (page 64)

[TriCore ISA 2008] Infineon Technologies AG: *TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1*. `http://www.infineon.com/dgdl/tc_v131_instructionset_v138.pdf?folderId=db3a304412b407950112b409b6cd0351&fileId=db3a304412b407950112b409b6dd0352`. January 2008. – Last Retrieved: April 2013 (page 16, 20, 25, 31, 133)

[Uhrig et al. 2005] Uhrig, Sascha ; Maier, Stefan ; Ungerer, Theo: Toward a Processor Core for Real-Time Capable Autonomic Systems. In: *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*. Los Alamitos, CA, USA : IEEE Computer Society, 2005, p. 19–22. – ISBN 0-7803-9313-9 (page 17)

[Ungerer 1993] Ungerer, Theo: *Datenflußrechner*. First Edition. Stuttgart, Germany : B.G. Teubner, 1993. – ISBN 3-519-02128-5 (page 17, 18, 100)

[Ungerer 1997] Ungerer, Theo: *Parallelrechner und parallele Programmierung*. First Edition. Heidelberg, Germany : Spektrum Akademischer Verlag GmbH, 1997. – ISBN 3-8274-0231-X (page 7, 8, 11, 13, 17, 22, 47, 117)

[Ungerer et al. 2010] Ungerer, Theo ; Cazorla, Francisco J. ; Sainrat, Pascal ; Bernat, Guillem ; Petrov, Zlatko ; Rochange, Christine ; Quiñones, Eduardo ; Gerdes, Mike ; Paolieri, Marco ; Wolf, Julian ; Cassé, Hugues ; Uhrig, Sascha ; Guliashvili, Irakli ; Houston, Michael ; Kluge, Florian ; Metzlaff, Stefan ; Mische, Jörg: MERASA: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. In: *IEEE Micro* 30 (2010), p. 66–75. – DOI http://dx.doi.org/10.1109/MM.2010.78. – ISSN 0272-1732 (page 2, 16, 19, 20, 23, 26, 45, 57, 58, 62, 86, 89, 106)

[V850E2/MN4 2013] Renesas Electronics Corporation: *V850E2/MN4 User's Manual Hardware Revision 3.0*. http://www.renesas.com/products/mpumcu/v850/V850e2mx/v850e2mx4/Documentation.jsp. February 2013. – Last Retrieved: April 2013 (page 17, 19)

[Valois 1995] Valois, John D.: Lock-Free Linked Lists Using Compare-and-Swap. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. New York, NY, USA : ACM, 1995 (PODC '95), p. 214–222. – DOI http://doi.acm.org/10.1145/224964.224988. – ISBN 0-89791-710-3 (page 31, 47, 48, 49)

[Valvano 2011] Valvano, Jonathan: *Embedded Microcomputer Systems: Real Time Interfacing*. International Edition of 3rd Revised Edition. Andover, MA, USA : Nelson Engineering Cengage Learning (EMEA), 2011. – ISBN 1-111-42626-0 (page 31)

[Vlissides et al. 1996] Vlissides, John ; Coplien, James ; Kerth, Norman L.: *Pattern Languages of Program Design*. Vol. 2. Boston, MA, USA : Addison-Wesley, 1996. – ISBN 0201895277 (page 116)

[Wandeler and Thiele 2006] Wandeler, Ernesto ; Thiele, Lothar: Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA : IEEE Press, 2006 (ASP-DAC '06), p. 479–484. – DOI http://dx.doi.org/10.1145/1118299.1118417. – ISBN 0-7803-9451-8 (page 63)

[Whitham and Audsley 2009] Whitham, Jack ; Audsley, Neil: Implementing Time-Predictable Load and Store Operations. In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. New York, NY, USA : ACM, 2009 (EMSOFT '09), p. 265–274. – DOI http://doi.acm.org/10.1145/1629335.1629371. – ISBN 978-1-60558-627-4 (page 25, 89)

[Wilhelm et al. 2008] Wilhelm, Reinhard ; Engblom, Jakob ; Aandreas, Ermedahl ; Holsti, Niklas ; Thesing, Stephan ; Whalley, David ; Bernat, Guillem ; Ferdinand, Christian ; Heckmann, Reinhold ; Mitra, Tulika ; Mueller, Frank ; Puaut, Isabelle ; Puschner, Peter ; Staschulat, Jan ; Stenström, Per: The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools. In: *ACM Transactions on Embedded Computing Systems* 7 (2008), May, No. 3, p. 36:1–36:53. – DOI http://doi.acm.org/10.1145/1347375.1347389. – ISSN 1539-9087 (page 1, 5, 6, 7, 55, 86)

[Wilhelm et al. 2009a] Wilhelm, Reinhard ; Ferdinand, Christian ; Cullmann, Christoph ; Grund, Daniel ; Reineke, Jan ; Triquet, Benoît: Designing Predictable Multicore Architectures for Avionics and Automotive Systems. In: *Workshop on Reconciling Performance with Predictability (RePP)*, October 2009 (page 2, 86)

[Wilhelm et al. 2009b] Wilhelm, Reinhard ; Grund, Daniel ; Reineke, Jan ; Schlickling, Marc ; Pister, Markus ; Ferdinand, Christian: Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (2009), July, No. 7, p. 966–978. – DOI http://dx.doi.org/10.1109/TCAD.2009.2013287. – ISSN 0278-0070 (page 2, 19, 86)

[Wilhelm et al. 2009c] Wilhelm, Reinhard ; Grund, Daniel ; Reineke, Jan ; Schlickling, Marc ; Pister, Markus ; Ferdinand, Christian: Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (2009), July, No. 7, p. 966–978. – DOI http://dx.doi.org/10.1109/TCAD.2009.2013287. – ISSN 0278-0070 (page 2, 55, 56, 57)

[Wind River Hypervisor] Wind River: *Wind River Hypervisor*. http://windriver.com/products/hypervisor/. – Last Retrieved: April 2013 (page 2)

[Wolf et al. 2010a] Wolf, Julian ; Gerdes, Mike ; Kluge, Florian ; Uhrig, Sascha ; Mische, Jörg ; Metzlaff, Stefan ; Rochange, Christine ; Cassé, Hugues ; Sainrat, Pascal ; Ungerer, Theo: RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor. In: *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Los Alamitos, CA, USA : IEEE Computer Society, 2010 (ISORC '10), p. 193–201. – DOI http://dx.doi.org/10.1109/ISORC.2010.31. – ISSN 1555-0885 (page 1, 24, 32, 57, 77, 88)

[Wolf et al. 2011] WOLF, Julian ; GERDES, Mike ; KLUGE, Florian ; UHRIG, Sascha ; MISCHE, Jörg ; METZLAFF, Stefan ; ROCHANGE, Christine ; CASSÉ, Hugues ; SAINRAT, Pascal ; UNGERER, Theo: RTOS Support for Execution of Parallelized Hard Real-Time Tasks on the MERASA Multi-Core Processor. In: *International Journal of Computer Systems, Science & Engineering (CSSE)* 26 (2011), November, No. 6. – ISSN 0267 6192 (page 24, 32, 57, 88)

[Wolf et al. 2010b] WOLF, Julian ; KLUGE, Florian ; GULIASHVILI, Irakli: Final System-Level Software for the MERASA Processor / Institute of Computer Science, University of Augsburg. October 2010 (2010-08). – Technical Report (page 21, 32)

[Xu et al. 1992] XU, Hong ; MCKINLEY, Philip K. ; NI, Lionel M.: Efficient Implementation of Barrier Synchronization in Wormhole-Routed Hypercube Multicomputers. In: *Proceedings of the 12th International Conference on Distributed Computing Systems*. Washington, DC, USA : IEEE Computer Society, June 1992 (DCS '92), p. 118–125. – DOI http://dx.doi.org/10.1109/ICDCS.1992.235048 (page 46)

[Yan and Zhang 2008] YAN, Jun ; ZHANG, Wei: WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In: *Proceedings of Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA : IEEE Computer Society, April 2008 (RTAS '08), p. 80 –89. – DOI http://dx.doi.org/10.1109/RTAS.2008.6. – ISSN 1080-1812 (page 2, 19, 86)

[Yoon et al. 2011] YOON, Man-Ki ; KIM, Jung-Eun ; SHA, Lui: Optimizing Tunable WCET with Shared Resource Allocation and Arbitration in Hard Real-Time Multicore Systems. In: *Proceedings of the 32nd IEEE Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, December 2011 (RTSS '11), p. 227–238. – DOI http://dx.doi.org/10.1109/RTSS.2011.28. – ISSN 1052-8725 (page 2, 86)

[Zhu et al. 2007] ZHU, Weirong ; SREEDHAR, Vugranam C. ; HU, Ziang ; GAO, Guang R.: Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In: *Proceedings of the 34th annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2007 (ISCA '07), p. 35–45. – DOI http://doi.acm.org/10.1145/1250662.1250668. – ISBN 978-1-59593-706-3 (page 101)

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

ACET average-case execution time, page 62, 71, 80
APD Activity Pattern Diagram, page 131
API application programming interface, page 22, 116

BCET best-case execution time, page 5, 53, 84
BIOS Basic Input Output System, page 22
BSP bulk synchronous parallel, page 87

CAN controller area network, page 125
CAS compare-and-swap, page 13, 15, 16, 18, 31, 45–50
CASN multi-word compare-and-swap, page 15, 50
CFG control flow graph, page 4, 55, 64, 66–73, 75, 76, 87, 169
CMP chip multiprocessors, page 46, 50, 55
COTS commercial off-the-shelf, page 1, 2, 16, 19, 21, 49, 64, 88
CPG concurrent program graph, page 87

D-ISP dynamic instruction scratchpad, page 20, 55, 64
DMP deterministic shared-memory multiprocessing, page 101
DRAM Dynamic Random-Access Memory, page 25
DSP data scratchpad, page 20, 55

ERG Exclusives Reservation Granule, page 30

F&A fetch-and-add, page 13, 14, 16, 25, 31, 44, 46, 47
F&D fetch-and-decrement, page 14, 25, 27, 28, 34, 38, 43, 58, 59, 68, 69, 73, 74, 76, 77, 79, 133, 169
F&I fetch-and-increment, page 13, 14, 25, 27–34, 36–38, 42–46, 58, 59, 68–70, 73–78, 80, 84, 110–112, 128, 129, 133–136, 169
F&I/F&D fetch-and-increment/fetch-and-decrement, page 14, 16, 26–28, 32, 34, 38, 40, 58, 59, 61, 63, 68–71, 73, 74, 77–80, 88, 104, 110, 134, 169
FCFS *first-come, first-served*, page 10, 26, 59
FFT Fast-Fourier-Transformation, page 80
FIFO *first in, first out*, page 10, 23, 26, 28–31, 34–36, 38, 43, 44, 48, 73–76, 79, 80, 90, 92, 97, 101, 133, 134
FPGA Field-Programmable Gate Array, page 20, 25, 58, 110

HRT hard real-time, page 1–5, 7, 10, 11, 13, 19–21, 23, 24, 27, 29, 31–38, 41–50, 53, 54, 58, 59, 63, 64, 71, 72, 74, 75, 78, 80, 87, 88, 90, 94, 101, 102, 114, 115, 118, 120, 131–138
HTM hardware transactional memory, page 49, 50

I/O Input/Output, page 24, 53, 55, 125
ILP Integer Linear Programming, page 6, 131

ISA instruction set architecture, page 13, 15–17, 24, 25, 27–29, 31, 47, 55, 66, 100, 120–123, 127–129, 133
ISR interrupt service routine, page 54

LL/SC load-linked/store-conditional, page 15, 16, 18, 27, 30, 31, 48, 49, 100

MERASA *Multi-core Execution of Hard Real-time Applications Supporting Analysability*, page 16, 20, 21, 23, 25–29, 31–34, 45, 47, 54, 55, 57–61, 63, 64, 66–68, 77–79, 82–84, 86, 89–91, 94, 96, 97, 101, 102, 104–106, 110–112, 123, 125, 127–129, 133–135, 139–143, 146, 148, 150
MOET maximum observed execution time, page 5, 7
MPSoC multi-processor system-on-chip, page 131

NHRT non-hard real-time, page 5, 20, 44, 47, 59, 63, 71, 78, 102, 134
NoC Network-on-Chip, page 55, 100
NRT non-real-time, page 5

OPL Our Pattern Language, page 118
OS Operating System, page 11, 21, 22, 48, 51, 116
OTAWA *Open Toolbox for Adaptive WCET Analysis*, page 7, 20, 53, 55, 64, 66, 68, 70, 77, 80, 82, 84, 106, 110, 113, 127, 129, 133, 136, 169

PAR Preemtable Atomic Regions, page 50
parMERASA *Multi-Core Execution of parallelised Hard Real-Time Applications Supporting Analysability*, page 31, 54, 57, 118, 121, 124, 127, 129, 131, 132, 137, 138
PRET precision timed, page 87
PWM pulse-width modulation, page 125

RCDC relaxed consistency deterministic computer, page 101
RISC Reduced Instruction Set Computing, page 15, 16
RMW read-modify-write, page 3, 4, 7, 13–19, 22–27, 29, 31, 33, 34, 40, 45, 53, 54, 58–60, 63, 68, 77–79, 89–97, 100–107, 109, 112, 114, 128, 133–136
RPC remote procedure call, page 50
RSF Request-Store-Forward, page 101
RTOS Real-Time Operating System, page 1, 11, 12, 21–23, 27–29, 32, 34, 71, 77, 120–123, 128, 129, 139–143, 146, 148, 150
RTTM real-time transactional memory, page 50

SB Synchronisation-operation Buffer, page 100
SCC Single-chip Cloud Computer, page 17
SDRAM Synchronous Dynamic Random-Access Memory, page 25, 26, 92, 93, 95, 110
SMP symmetric multiprocessing, page 88
SMT simultaneous multithreading, page 20, 47, 55, 59, 63, 134
SRT soft real-time, page 5, 48–50
SSB Synchronisation State Buffer, page 101
STM software transactional memory, page 49–51

TAS test-and-set, page 11, 13, 14, 16, 17, 25–27, 32–38, 40, 45, 47, 48, 58, 59, 61, 63, 67–69, 71, 78, 79, 88, 92, 95, 104, 110, 130, 133, 134, 169

TCB thread control block, page 21

TDMA time-division multiple access, page 58, 62, 63, 86

UML Unified Modeling Language, page 131, 138

WCET worst-case execution time, page 1, 3–7, 10, 12, 15, 17, 19–21, 24, 29, 33, 38, 40, 42, 44, 45, 53–55, 57–64, 66–80, 82–84, 86–92, 100, 101, 104–106, 110–115, 118, 120–125, 127–129, 131, 133–137, 169

WCML worst-case memory latency, page 3, 19, 24, 33, 47, 53–55, 58–61, 63, 64, 68, 69, 77, 78, 80, 82, 86, 89–91, 95, 101–109, 134–136

WCRT worst-case response time, page 50, 87

WCWT worst-case waiting time, page 55, 57, 67, 68, 71, 73, 74, 78, 88, 110

Mike Gerdes

University of Augsburg
Department of Computer Science
Systems and Networking
Universitätsstr. 6a, 86135 Augsburg

Personal

Born on February 7, 1978.

German Citizen.

Education

Graduation in computer sciences (degree: Diplom-Informatiker),
University of Augsburg, 2008.

Ph.D. candidate (Computer Science), University of Augsburg, 2013.

Employment

Officer at German Air Force 1997–2003.

Researcher at University of Augsburg, Dep. of Computer Science, 2008–

Publications

Mike Gerdes, Florian Kluge, Christine Rochange, and Theo Ungerer: The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallel HRT Programs. *In:* Proc. of Embedded and Real-Time Computing Systems and Applications (RTCSA'12), p. 88–97, 2012.

Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat: Time Analysable Synchronisation Techniques for Parallelised HRT Applications. *In:* Proc. of Design, Automation and Test in Europe (DATE'12), p. 671–676, 2012.

Mike Gerdes, Julian Wolf, Irakli Guliashvili, Theo Ungerer, Michael Houston, Guillem Bernat, Stefan Schnitzler, and Hans Regler: Large Drilling Machine Control Code – Parallelisation and WCET Speedup. *In:* Proc. of 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), p. 91–94, 2011.

Mike Gerdes, Julian Wolf, Ji Zhang, Sascha Uhrig, and Theo Ungerer: Multi-Core Architectures for Hard Real-Time Applications. *In:* 4th Int'l Summer School on ACACES, Poster Abstracts, 2008.

Further Publications

Florian Kluge, Mike Gerdes, and Theo Ungerer: An Operating System for Safety-Critical Applications on Manycore Processors. *submitted for publication*, 2013.

Ralf Jahr, Mike Gerdes, and Theo Ungerer: An Approach for Parallelization with Parallel Design Patterns. *submitted for publication*, 2013.

Ralf Jahr, Mike Gerdes, and Theo Ungerer: A Pattern-supported Parallelization Approach. *In:* Proc. of the 2013 Int'l Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), February 2013.

Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla: A Hard Real-Time Capable Multi-Core SMT Processor. *In:* ACM Transactions on Embedded Computing Systems (TECS), Vol. 12 No. 3, March, 2013.

Florian Kluge, Mike Gerdes and Theo Ungerer: AUTOSAR OS on a Message-Passing Multicore Processor. *In:* Proc. of 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), 2012.

Julian Wolf, Mike Gerdes, Florian Kluge, Sascha Uhrig, Jörg Mische, Stefan Metzlaff, Christine Rochange, Hugues Cassé, Pascal Sainrat, and Theo Ungerer: RTOS Support for Execution of Parallelized Hard Real-Time Tasks on the MERASA Multi-Core Processor. *In:* Int'l Journal of Computer Systems, Science&Engineering (CSSE), ISSN 0267 6192, Vol. 26, No. 6, 2011.

Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Hugues Cassé, Christine Rochange, Eduardo Quiñones, Sascha Uhrig, Mike Gerdes, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, Jörg Mische, Marco Paolieri, Julian Wolf: MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability. *In:* IEEE Micro 2010, Special Issue on European Multicore Processing Projects, Vol. 30 No. 5, Sept./Oct. 2010.

Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu: WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-core. *In:* 10th Int'l Workshop on Worst-Case Execution-Time Analysis (WCET'10) in conjunction with the 22nd Euromicro Int'l Conference on Real-Time Systems, 2010.

Julian Wolf, Mike Gerdes, Florian Kluge, Sascha Uhrig, Jörg Mische, Stefan Metzlaff, Christine Rochange, Hugues Cassé, Pascal Sainrat, and Theo Ungerer: RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. *In:* Proc. of IEEE Int'l Symp. on Object/component/service-oriented Real-time distributed Computing (ISORC), 2010.

Wolfgang Trumler and Mike Gerdes: Towards an automated detection of self-organizing behavior. *In:* INFORMATIK 2008, Beherrschbare Systeme - dank Informatik, Band 2, 38. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2008.

Last updated: July 22, 2013

(see also: `http://scholar.google.de/citations?user=jrOkK2MAAAAJ`)