Deterministic execution of multithreaded applications for reliability of multicore systems

DETERMINISTIC EXECUTION OF MULTITHREADED APPLICATIONS FOR RELIABILITY OF MULTICORE SYSTEMS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 19 juni 2015 om 10:00 uur

door

Hamid MUSHTAQ

Master of Science in Embedded Systems
geboren te Karachi, Pakistan

Keywords: Multicore, Fault Tolerance, Reliability, Deterministic Execution, WCET

The work in this thesis was supported by Artemis through the SMECY project (grant 100230).

Cover image: An Artist's impression of a view of Saturn from its moon Titan. The image was taken from http://www.istockphoto.com and used with permission.

ISBN 978-94-6186-487-1

Dedicated to my family

ACKNOWLEDGEMENTS

I am thankful to Almighty Allah, who in his infinite mercy has guided me to complete my PhD. My deepest gratitude is for my supervisors Dr. Zaid Al-Ars and Professor Koen Bertels. Their professional guidance, enthusiastic encouragement, valuable support, useful critiques and constructive recommendations were the key motivations throughout my PhD. I have been amazingly fortunate to have an advisor like Dr. Zaid Al-Ars, who gave me the freedom to explore on my own and at the same time guided me whenever I faltered. I am thankful to him for carefully reading and commenting on countless revisions of research papers and encouraging me to meet the high standards.

I am especially indebted to my beloved wife and children: Ismail and Khadija, for their unflagging support, strength and encouragement throughout my PhD and for understanding the importance of this work and its hectic working hours.

I would also like to thank the CE staff, which includes CE secretary Lidwina Tromp and system administrators Eef and Erik for their administrative and technical assistance throughout.

Moreover, I take this opportunity to thank my fellow PhD students and friends, such as, Imran Ashraf, Nauman Ahmed, Mottaqiullah Taouil, Innocent Agbo, George Razvan Voicu, Cuong Pham, Tariq Abdullah, Olivier Debant, Muhammad Nadeem, Faisal Nadeem, Seyab Khan, Fakhar Anjam, AN Tabish, Shah Muhammad, Shehzad Gishkori, Faisal Aslam, Laiq Hassan, Atif Khan and many more. They all helped make my time during PhD more fun and interesting.

Furthermore, a special thanks to my family, for encouraging me in all my pursuits and inspiring me to follow my dreams. I have no words to express how grateful I am to my parents for all of the sacrifices that you have made on my behalf. Your prayers and support always benefited me. I would also like to thank my brothers Ishtiaq and Aleem and my sister Asma for their love and support.

I am also highly obliged to my father-in-law and my mother-in-law for their encouragement and support. I am especially thankful to my mother-in-law for coming here and helping us during the birth of my daughter Khadija.

Finally, I would like to dedicate this thesis to all my teachers, professors, friends, family and so many others whom I have inadvertently left out. I sincerely thank all of them for their help, support and guidance.

Hamid Mushtaq
Delft, June 2015

SUMMARY

Constant reduction in the size of transistors has made it possible to implement many cores on a single die. However, smaller transistors are more susceptible to both temporary and permanent faults. To make such systems more reliable, online fault tolerance techniques can be applied. A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the program replication approach. In this approach, the replicated copies of a program (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the replicas perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

In this thesis, we employed two techniques for doing so, which are record/replay and deterministic multithreading. Both of our schemes are implemented using a user-level library and do not require a modified kernel. Moreover, they are very portable since they do not depend upon any special hardware for deterministic execution. In addition, we compare the advantages and disadvantages of both schemes in terms of performance, memory consumption and reliability. We also showed how our techniques improve upon existing techniques in terms of performance, scalability and portability. Lastly, we implemented specialized hardware extensions to further improve the performance and scalability of deterministic multithreading.

Deterministic multithreading is useful not only for fault tolerance, but also for debugging and testing of multithreaded applications running on a multicore system. It can be useful in reducing the time needed to calculate the worst-case-execution-time (WCET) of tasks running on multicore systems, as deterministic multithreading reduces the possible number of states a multithreaded program can reach. Finding a good WCET estimate (less pessimistic) of a real time task is much simpler if it runs on a single core processor than if it runs on a multicore processor concurrently with other tasks. This is because those tasks can share resources, such as a shared cache or a shared bus, and/or may need to concurrently read and/or write shared data. In this thesis, we show that using deterministic shared memory accesses helps in reducing the possible number of states used by the estimation algorithm and therefore reduce the WCET calculation time. Moreover, we implemented optimizations to further reduce WCET calculation time as well as to get a tighter WCET estimate, besides utilizing our specialized hardware extensions for that purpose.

SAMENVATTING

Door de voortdurende verkleining van transistoren is het nu mogelijk geworden om meerdere processoren op een chip te plaatsen. Echter kleinere transistoren zijn gevoeliger voor zowel tijdelijke en permanente fouten. Om dergelijke systemen betrouwbaarder te maken kunnen online fouttolerantie technieken worden toegepast. Een mogelijke manier om fouttolerantie te implementeren is door het repliceren van een programma. In deze benadering worden de gerepliceerde kopieën van een programma (zogenaamde replica's) op dezelfde manier uitgevoerd waarmee ze dezelfde output kunnen produceren. Om dit mogelijk te maken zouden wij asynchrone signalen en niet-deterministische functies deterministisch moeten maken. Dit wordt meestal gedaan door het opnemen de niet-deterministische gebeurtenissen bij de ene replica en daarna het afspelen van deze opgenomen gebeurtenissen bij de andere replica's.

In dit proefschrift hebben we gebruik gemaakt van twee technieken om deterministisch gedrag te garanderen: de record/replay en de deterministische multithreading. Allebei technieken zijn zeer draagbaar aangezien dat zij geen besturingssysteem kernel veranderingen of speciale hardware nodig hebben. Bovendien hebben wij in dit proefschrift de voor en nadelen van beide technieken vergeleken in termen van snelheid, geheugengebruik en betrouwbaarheid. We toonden ook aan hoe onze technieken verbeteren op bestaande technieken in termen van snelheid, schaalbaarheid en draagbaarheid. Ten slotte, hebben we een gespecialiseerde hardware-uitbreidingen voor deterministische multithreading geïmplementeerd om de snelheid en schaalbaarheid te verbeteren.

Deterministische multithreading is niet alleen nuttig voor fouttolerantie, maar ook voor het testen van multithreaded applicaties die op een multicore systeem draaien. Dit kan nuttig zijn bij het verminderen van de tijd die nodig is om de worst-case-execution-time (WCET) te berekenen van programma's die op multicore systemen draaien. Het berekenen van een goede WCET benadering is veel eenvoudiger als het programma op een enkele processor draait ten opzichte van meerdere processoren tegelijk. In dit proefschrift tonen wij dat het gebruik van een deterministisch geheugen model de complexiteit van het berekenen van WCET aanzienlijk kan verminderen. Bovendien hebben wij meerdere optimalisaties geïmplementeerd om de complexiteit nog verder te reduceren.

CONTENTS

1 INTRODUCTION

While modern nano-scale technology has made it possible to implement multiple cores on a single die, it has also aggravated the reliability problem, as smaller transistors are more prone to permanent and transient faults. However, online fault tolerance techniques can mitigate errors in such systems. For that purpose, this thesis employed two techniques, record/replay and deterministic multithreading. Besides fault tolerance, deterministic multithreading can also help in reducing the calculation time for the worst-case-execution-time (WCET) of real time systems running on shared memory multicore systems. Therefore, in this thesis, we also tested the improvement in calculation time deterministic multithreading brings.

In this chapter, Section 1.1 discusses the basic concepts of fault tolerance and WCET. This is followed by Section 1.2 on background and related work. Next, we define our contributions in Section 1.3. Finally, we describe the thesis organization in Section 1.4.

1.1. BASIC CONCEPTS

In this section, we present the basic concepts related to the field of fault tolerance and WCET. Our discussion is based on the way a system behaves and interacts with other systems in its environment [2].

In this thesis, a system can be hardware based, for example a processor, or software based, such as a running application. A system consists of components which can be systems themselves. The service delivered by a system is its behavior as perceived by other systems using it. The total state of a system is the set of its internal and external states. The external state of a system is represented by its output. A system is said to fail when its external state deviates from the correct state. The cause of this failure is **fault**(s) within the system or external to it. Fault propagation is illustrated in Figure 1.1. When a fault becomes **active**, it would impact the total state of one or more components of the system. The deviation of the total state of a component from the correct state is known as an **error**. When an error propagates to affect the external state of the system, the error is said to be activated. When the error is activated, **failure** of the system is said to have

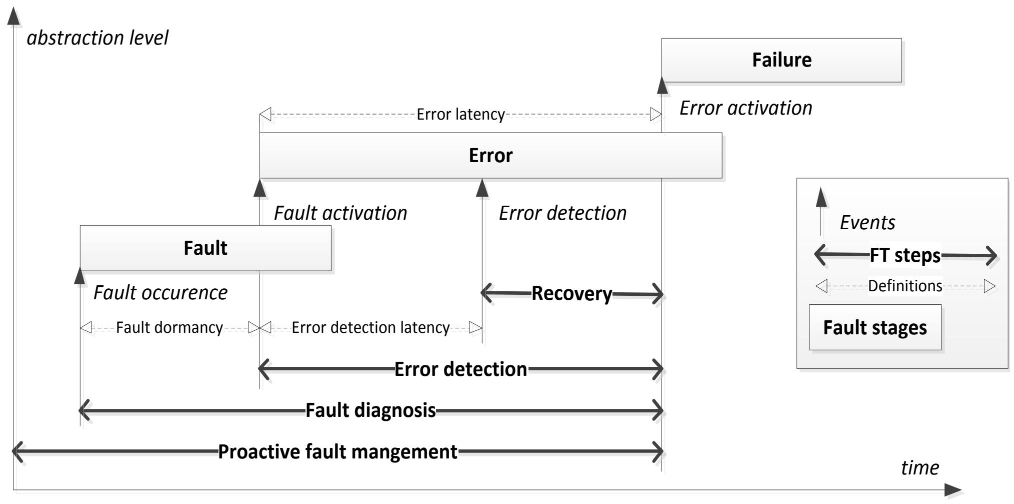

Figure 1.1: Fault propagation and fault tolerance

occurred. The time between **fault activation** and failure is known as **error latency**. In other words, a fault might lead to an error which in turn might lead to the failure of the system.

FAULTS

Faults can be classified into four different classes depending on their *persistence*, *effect*, *source* and *boundary*.

With respect to persistence, a fault can be **permanent**, **intermittent** or **transient**. Permanent faults are continuous in time, while a transient fault occurs for only a short period of time. An intermittent fault is a repetitive malfunction of a device or system that occurs at intervals.

With respect to effect, a fault can be either **activated** or **dormant**. An activated fault is one which has produced an error, while a dormant fault is one that has yet to produce an error. An activated fault can be further classified into **latent** and **detected**, where a latent fault is one which has produced an error that has still not been detected by the system.

The source of a fault can be either **software** or **hardware**. Software faults can be for example design faults or malicious attacks like trojan horses.

Lastly, a fault can be either due to a component **internal** to the system or **external** to it.

A fault can produce a number of errors in a computing system, such as, control-flow errors, data corruption errors, logical errors, buffer overflows, memory leaks, data races, deadlocks/livelocks, infinite loops and wild memory writes, etc.

FAILURES

Failures can be classified into three different classes based on their *domain*, *action* and *consistency*.

In terms of domain, failures can be either **timing** related or **content** related. Timing failures mean that the failing system either responds too early or too late. On the other hand, content failures mean that the content of the information delivered by the system is in corrupt state.

In terms of action taken by a failing system, failures can be divided into **halt** and **erratic** failures. By halt failure, we mean that the system stops responding on failure, while erratic failure means that the failing system keeps responding but in an abnormal manner. Halting on failure is a good property, as errors are not propagated to other systems in the environment. Systems which halt on failure, are known as **fail-stop** systems.

In terms of **consistency**, there can be **byzantine** and **consistent** failures. When a byzantine failure happens, some or all users of the system will perceive different service. On the other hand, for consistent failures, all users will perceive identical service.

Service failure of a system causes a permanent or transient external fault for other system(s) that receive service from that system.

FAULT TOLERANCE

Fault tolerance means to avoid failures in the presence of faults. A system is said to be **fault tolerant** if faults do not affect the external state of that system. It can however allow its components to fail, as long as they do not corrupt its external state. A fault tolerant system must be able to detect errors and recover from them. The time between fault activation and error detection is known as **error detection latency**.

Figure 1.2 shows the steps which are taken to make a system fault tolerant. These steps are **proactive fault management**, **error detection**, **fault diagnosis** and **recovery**. More details about these steps are discussed in Chapter 2.

WORST CASE EXECUTION TIME (WCET)

Adapting multicore systems to real time embedded systems is a challenging task, as a real time process, besides being error free, must also meet timing deadlines. The real time scheduler needs to know the maximum time in which a task would complete. This time is known as the WCET. Finding a good WCET estimate (less pessimistic) of a task is much simpler if it runs on a single core processor than if it runs on a multicore processor concurrently with other tasks. This is because those tasks can share resources, such as shared caches or a shared bus, and/or may need to concurrently read and/or write shared data.

1.2. BACKGROUND AND RELATED WORK

1.2.1. FAULT TOLERANCE

A fault tolerant system which uses redundant execution needs to make sure that the redundant processes do not diverge in the absence of faults, that is, they should have the same states for the same input. In a single threaded program, in the absence of any fault, the only possible causes of divergence among the replicas can be non-deterministic

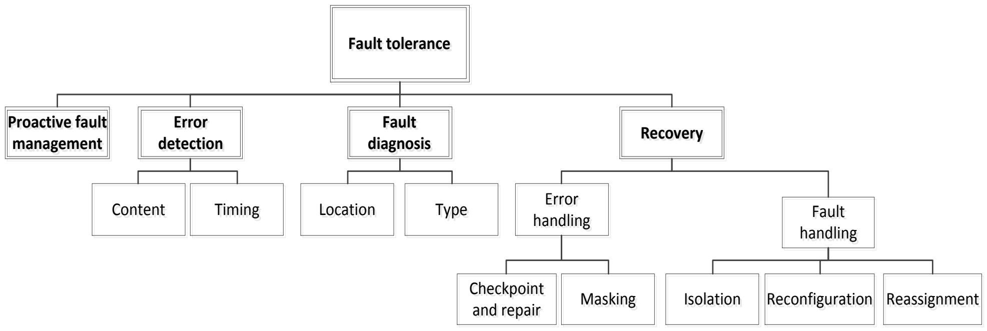

Figure 1.2: Classification of steps used for fault tolerance

functions (such as *gettimeofday*) or asynchronous signals/interrupts.

However, in multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses. These accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve.

One method to ensure redundant processes access shared memory in the same order is record/replay. In this method, all interleavings of shared memory accesses by different cores or processors are recorded in a log, which can be replayed to have a replica which follows the original execution. Examples of schemes using this method are Rerun [3] and Karma [4]. These schemes intercept cache coherence protocols to record inter-processor data dependencies, so that they can be replayed later on, in the same order. While Rerun only optimizes recording, Karma optimizes both recording and replaying, thus making it suitable for online fault tolerance. It shows good scalability as well. One disadvantage of record/replay approaches is that they require a large memory for recording. Moreover, when used for fault tolerance, the redundant processes need to communicate with each other, as one replica records the log while the other reads from it. Respec [5] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rolls-back and re-executes from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed, which can induce a large overhead.

The disadvantage of employing record/replay for deterministic shared memory accesses is that it requires communication between the replicas for shared memory accesses, making the fault tolerant method less reliable as the shared buffer used for communication can itself become corrupted by one of the replicas. Moreover, it requires extra memory.

To eliminate this communication and memory requirement, we can employ deterministic multithreading, where a multithreaded process has always the same memory interleaving for the same input. The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions, that is, provide strong determinism. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [6]. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [7]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. An example of such approach is Calvin [8], which uses the same concept as CoreDet for deterministic execution but makes use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, one can relax the requirements to improve efficiency. For example, Kendo [9] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without race conditions. The authors of Kendo call it weak determinism. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [1], weak determinism is sufficient for most well written multithreaded programs.

The basic idea of Kendo is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, Kendo still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counters that are deterministic [10]. Secondly, Kendo needs modification of the kernel to allow reading from the hardware performance counters.

More detailed survey on the fault tolerance techniques for shared memory multicore systems is given in Chapter 2.

1.2.2. WCET CALCULATION

Modern processors have features such as cache hierarchies and out of order execution, which are meant to improve the average case time of programs running on them. However, these features make it much more difficult to determine a tight WCET. In addition, more complex architectures mean more states for a model checker to keep track of, making it more prone to state explosion problems. Despite these problems, there exist production level tools, such as chronos [11], that can guess a good WCET for programs running on single core processors. Multicore systems on the other hand have an additional complexity, due to shared resources, such as shared memories. With shared memory, tasks running on different cores also need to synchronize to accesses the shared data, for example by using locks. This makes it difficult to deduce tight WCET bounds for such systems. Synchronization of shared memory accesses also means many different possible interleaving of the threads, which further aggravate the problem of calculating the WCET. They can have timing anomalies due to shared resources and shared memory accesses. For example, assume that a path ABD is the worst-case path if seen separately, where A, B and D are basic blocks. In the presence of shared L2 cache however, another path, say ACD might become the worst-case path if a thread running on another core evicts more instructions from C than from B in the L2 cache. Therefore, whenever analyzing WCET for a multicore, we always need to consider all the tasks running on different cores together, which can significantly increase the complexity of timing analysis.

Recently, there has been several papers published which deal with calculating WCET on multicore processors. A survey of those techniques is given in [12]. Some of those assume that there are no shared memory accesses by the tasks running on the different cores. In other words, they assume that tasks are running embarrassingly parallel to each other. They only cater for the problem of shared L2 cache accesses [13][14] and the shared bus [15]. Papers like [16] and [17] do consider shared memory synchronization, but they assume simpler processor architectures which do not have any cache, but only scratchpad memories. Such kind of processors are not mainstream and require special programming techniques, since the scratchpad memories have to be manually managed by the programmer.

[18] considers both cache coherence as well as synchronization operations such as spin locks for shared memory accesses. The authors use UPPAAL [19] based model checking for that purpose. They do take into account shared memory accesses, but their solution suffers from state explosion problem even for very simple programs. [22] also uses model checking but does not support synchronization operations. [20] recently proposed a mathematical model to determine WCET of multicore systems with caches and cache coherence using abstract interpretation. However, they still do not consider cache coherence that is generated due to accessing the shared synchronizing objects. Moreover, they do not perform any evaluation.

More detailed survey on the WCET calculation techniques for shared memory multicore systems is given in Chapter 6.

1.3. OUR CONTRIBUTION

1.3.1. FAULT TOLERANCE FOR MULTICORE SYSTEMS

A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the state machine replication approach [21]. In this approach the replicated copies of a process (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions (such as *gettimeofday*) deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the replicas perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

One way of making sure that the redundant processes access the shared memory in the same order is to perform record/replay where the leader process records the order of locks (to access shared memory) in a queue which is shared between the leader and follower. The follower in turn reads from that queue to have the same lock acquisitions order. This approach is used by Respec [5]. This is the first approach that we have used in this thesis. The second approach that we use is deterministic multithreading, where given the same input, a multithreaded process always has the same lock interleaving. This makes sure that the redundant processes acquire the locks in the same order without communicating with each other. We adapt the method used by Kendo [9] to do this, but unlike Kendo, our scheme neither requires deterministic hardware performance counters (which are not available on many platforms [10], including many x86 systems), nor kernel modification for deterministic execution. The logical clocks used for deterministic execution are inserted by the compiler instead.

Moreover, we also implemented hardware extensions to aid in deterministic multithreading. Having hardware decreases portability, but gives significant improvement in performance and scalability.

We can sum up our contributions to fault tolerance in this thesis as follows.

1. We discuss the implementation of our two schemes for deterministic execution on multicore platforms for fault tolerance.

2. Both of our schemes are implemented using a user-level library and do not require a modified kernel.

3. Both of our schemes are very portable since they do not depend upon any special hardware for deterministic execution.

4. Both of our schemes show better performance than existing approaches on selected benchmarks.

5. We compare the advantages and disadvantages of both schemes in terms of performance, memory consumption and reliability.

6. We also implement specialized hardware extensions for deterministic multithreading to improve the performance and scalability.

1.3.2. WCET CALCULATION

Real time tasks, besides being error free, must also meet timing deadlines. For that purpose, it is necessary to calculate the WCET of a real time task. Finding a good WCET estimate (less pessimistic) of a real time task is much simpler if it runs on a single core processor than if it runs on a multicore processor concurrently with other tasks. This is because those tasks can share resources, such as shared cache or shared bus, and/or may need to concurrently read and/or write shared data.

Recently, there has been an increase in interest to solve the problem of finding WCET for tasks running on multicore processors, from on-chip hardware support to software solutions for commodity off the shelf (COTS) processors. But most of those do not take into account the shared memory accesses. In [18], the authors do take into account shared memory accesses, but the state explosion problem of the model checking based approach they use limits the effectiveness of that approach.

In this thesis, we show that using deterministic shared memory accesses [9] [25] helps reducing the possible number of states used by the model checker and therefore reduce the WCET calculation time. We can sum up the contribution of this thesis to WCET calculation as follows.

1. Limiting the state space explosion problem by utilizing deterministic execution when calculating the WCET of a multithreaded program running on multicores using model checking.

2. Implementing optimizations to further reduce WCET calculation time as well as to get a tighter WCET estimation.

3. We also utilized our specialized hardware to further reduce the WCET calculation time.

1.4. THESIS ORGANIZATION

The thesis is organized as follows.

In **Chapter 2**, we give a survey of fault tolerance techniques for shared memory multicore systems

In **Chapter 3**, we discuss our own implementation of fault tolerance for shared memory multicore systems using record/replay. We also compare the performance with the state of the art.

In **Chapter 4**, we discuss our own implementation of deterministic multithreading, which like record/replay, can be used for deterministic execution of redundant processes running on multicore systems for fault tolerance. We also compare our method with the state of the art.

In **Chapter 5**, we give a comparison of fault tolerance using record/replay and deterministic multithreading. It also discusses the performance improvement after using hardware based deterministic multithreading.

In **Chapter 6**, we show reduction in WCET calculation time that we get by using deterministic multithreading.

In **Chapter 7**, we present the conclusions and discuss future research areas.

REFERENCES

[1] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. PLDI '07, pages 89–100.

[2] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, pages 11 – 33, 2004.

[3] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.

[4] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.

[5] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, ASPLOS '10, pages 77–90.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[7] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. SOSP '11, pages 327–336.

[8] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. HPCA '11, pages 333 –334, feb.

[9] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. ASPLOS '09, pages 97–108.

[10] V. Weaver and J. Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? FHPM '10.

[11] Xianfeng Li, Yun Liang, Tulika Mitra, Abhik Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. Science of Computer Programming, Special issue on Experimental Software and Toolkit, 69(1-3), December 2007.

[12] Hamid Mushtaq and Zaid Al-Ars and Koen Bertels. Accurate and efficient identification of worst-case execution time for multicore processors: A survey In *IDT 2013*.

[13] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *RTAS*,2008.

[14] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multicore processors with shared direct-mapped instruction caches. In *RTCSA*, 2009.

[15] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, 2010.

[16] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET Analysis of Real-Time Parallel In *13th International Workshop on Worst-Case Execution Time Analysis*, 2013.

[17] Mike Gerdes, Theo Ungerer and Rudolf Knorr Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores Phd Thesis.

[18] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards wcet analysis of multicore architectures using uppaal. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[19] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[20] Sudipta Chattopadhyay. Time-predictable Execution of Embedded Software on Multi-core Platforms. Phd Thesis, National University of Singapore, 2012.

[21] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[22] Lan Wu and Wei Zhang. A model checking based approach to bounding worst-case execution time for multicore processors. *ACM Trans. Embed. Comput. Syst.*, 11(S2):56:1–56:19, August 2012.

[23] R. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, 1(1):17 –22, March 2001.

[24] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. ISCA '11, pages 201–212.

[25] H. Mushtaq, Z. Al-Ars, and K. Bertels. Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 721–730, 2012.

2

Fault tolerance techniques for multicore systems

SUMMARY
With the advent of modern nano-scale technology, it has become possible to implement multiple processing cores on a single die. The shrinking transistor sizes however have made reliability a concern for such systems as smaller transistors are more prone to permanent as well as transient faults. To reduce the probability of failures of such systems, online fault tolerance techniques can be applied. These techniques need to be efficient as they execute concurrently with applications running on such systems. This chapter discusses the challenges involved in online fault tolerance and existing work which tackles these challenges. We classify fault tolerance into four different steps which are proactive fault management, error detection, fault diagnosis and recovery and discuss related work for each step, with focus on techniques for shared memory multicore/multiprocessor systems. We also highlight the additional difficulties in tolerating faults for parallel execution on shared memory multicore/multiprocessor systems.

This chapter is based on the following paper.

Mushtaq, H.; Al-Ars, Z.; Bertels, K., *Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems*, Design and Test Workshop (IDT), 2011 IEEE 6th International, pp. 12-17, 11-14 Dec. 2011

Survey of Fault Tolerance Techniques for Shared Memory Multicore/Multiprocessor Systems

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

Abstract— **With the advent of modern nano-scale technology, it has become possible to implement multiple processing cores on a single die. The shrinking transistor sizes however have made reliability a concern for such systems as smaller transistors are more prone to permanent as well as transient faults. To reduce the probability of failures of such systems, online fault tolerance techniques can be applied. These techniques need to be efficient as they execute concurrently with applications running on such systems. This paper discusses the challenges involved in online fault tolerance and existing work which tackles these challenges. We classify fault tolerance into four different steps which are proactive fault management, error detection, fault diagnosis and recovery and discuss related work for each step, with focus on techniques for shared memory multicore/multiprocessor systems. We also highlight the additional difficulties in tolerating faults for parallel execution on shared memory multicore/multiprocessor systems.**

I. INTRODUCTION

It has become possible to integrate billions of transistors on a single die with modern nano-scale technology and therefore allow many processing cores to be implemented on the same chip. While this advancement allows software with a large level of parallelism to execute very efficiently on such processors, it has also introduced reliability issues as the small transistors are more susceptible to both transient [2] and permanent [18] faults. This necessitates the implementation of efficient and scalable online *fault tolerance* (FT) techniques to reduce the probability of failures of such systems.

Fault tolerance of programs running sequentially on uniprocessors is well understood and many efficient solutions exist for that purpose. On the other hand, programs running in parallel on shared memory multicore processors present a greater challenge due to shared memory accesses, which are a frequent source of non-determinism. This paper gives a survey of work done on fault tolerance with primary focus on work for shared memory multicore systems.

Section II, discusses the basic concepts of a system, faults, failures and fault tolerance. Then we classify fault tolerance into four different steps: *proactive fault management* (discussed in Section III), *error detection* (discussed in Section IV), *fault diagnosis* (discussed in Section V) and *recovery* (discussed in Section VI). Redundant execution for fault tolerance is discussed in Section VII. We finally conclude this paper with Section VIII.

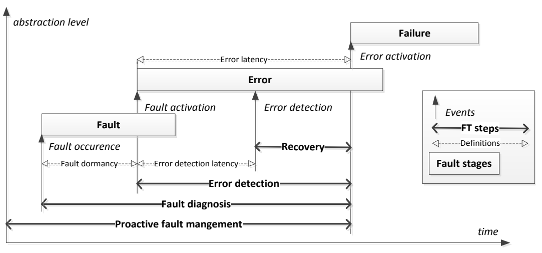

Fig. 1. Fault propagation and fault tolerance

II. BASIC CONCEPTS

In this section, we present the basic concepts related to the field of fault tolerance. Our discussion is based on the way a system behaves and interacts with other systems in its environment [38].

A *system* is an entity that interacts with other systems. A system can be hardware based, for example a processor, or software based, such as a running application. A system consists of components which can be systems themselves. The service delivered by a system is its behavior as perceived by other systems using it. The total state of a system is the set of its internal and external states. A system is said to fail when its external state deviates from the correct state. The cause of this failure is **fault**(s) within the system or external to it. Fault propagation is illustrated in Figure 1. When a fault becomes **active**, it would impact the total state of one or more components of the system. The deviation of the total state of a component from the correct state is known as an **error**. When an error propagates to affect the external state of the system, the error is said to be activated. When the error is activated, **failure** of the system is said to have occurred. The time between **fault activation** and failure is known as **error latency**. In other words, a fault might lead to an error which in turn might lead to the failure of the system.

A. Faults

Faults can be classified into four different classes depending on their *pesistence*, *effect*, *source* and *boundary*.

With respect to persistence, a fault can be **permanent**, **intermittent** or **transient**. Permanent faults are continuous in time, while a transient fault is random and occurs for only a short period of time. An intermittent fault is a repetitive malfunction of a device or system that occurs at intervals.

With respect to effect, a fault can be either **activated** or **dormant**. An activated fault is one which has produced an error, while a dormant fault is one that has yet to produce an error. An activated fault can be further classified into **latent** and **detected**, where a latent fault is one which has produced an error that has still not been detected by the system.

The source of a fault can be either **software** or **hardware**. Software faults can be for example design faults or malicious attacks like trojan horses.

Lastly, a fault can be either due to a component **internal** to the system or **external** to it.

A fault can produce a number of errors in a computing system, such as, control-flow errors, data corruption errors, logical errors, buffer overflows, memory leaks, data races, deadlocks/livelocks, infinite loops and wild memory writes etc.

B. Failures

Failures can be classified into three different classes based on their *domain*, *action* and *consistency*.

In terms of domain, failures can be either **timing** related or **content** related. Timing failures mean that the failing system either responds too early or too late. On the other hand, content failures mean that the content of the information delivered by the system is in corrupt state.

In terms of action taken by a failing system, failures can be divided into **halt** and **erratic** failures. By halt failure, we mean that the system stops responding on failure, while erratic failure means that the failing system keeps responding but in an abnormal manner. Halting on failure is a good property, as errors are not propagated to other systems in the environment. Systems which halt on failure, are known as **fail-stop** systems.

In terms of **consistency**, there can be **byzantine** and **consistent** failures. When a byzantine failure happens, some or all users of the system will perceive different service. On the other hand, for consistent failures, all users will perceive identical service.

Service failure of a system causes a permanent or transient external fault for other system(s) that receive service from that system.

C. Fault tolerance

Fault tolerance means to avoid failures in the presence of faults. A system is said to be **fault tolerant** if faults do not affect the external state of that system. It can however allow its components to fail, as long as they do not corrupt its external state. A fault tolerant system must be able to detect errors and recover from them. The time between fault activation and error detection is known as **error detection latency**.

Figure 2 shows the steps which are taken to make a system fault tolerant. These steps are **proactive fault management**, **error detection**, **fault diagnosis** and **recovery**. In coming sections, we discuss these steps and the related work done with special focus on shared memory multicore/multiprocessor systems.

III. Proactive fault management

Proactive fault management means predicting failures of components before they happen and taking precautionary steps to prevent them, as illustrated in Figure 1.

Software rejuvenation [11] is a proactive fault management technique that tries to avoid faults due to software aging. Software aging is the degradation of an application or system with time. Degradation can happen due to resource leakage, such as memory leaks or accumulation of numerical errors for example. In multithreaded applications, deadlocks may also appear due to software aging. Software rejuvenation tends to avoid these aforementioned problems by periodically restarting applications in a clean state.

Another way of performing Proactive fault management is to proactively check for errors in the system, that is check for errors when system is idle for example or by doing system monitoring. One such technique is memory scrubbing [21] in which memories are periodically checked for errors and corrected, even while they are not in use. Another example of a system which uses proactive error checking is [23] which predicts faults through system monitoring. In case of abnormal behavior detection, such as aberrant temperature or disk errors in a node, the tasks executing on it are migrated to a healthy node.

IV. Error detection

Error detection is the process of detecting errors in the system. Timing errors are normally detected by using watchdog timers, while for content errors, redundant execution is normally applied.

A. Watchdog timers and processors

A watchdog timer is a timer that is used to check if a system or a subsystem in it, is stuck, for example due to an infinite loop, in which case it triggers a corrective measure by the system.

A watchdog processor is a coprocessor that is used to detect system level errors by monitoring the behavior of the system. A survey of different kinds of watchdog processors is given in [16]. Watchdog processors can be used to check control flow errors. This is done by associating signatures at each node of a program and providing same signatures to the watchdog processor. [15] shows that 90 percent of control flow errors can be detected through watchdog processors with very low hardware and memory overhead.

B. Redundancy

Redundancy is a technique in which multiple processing elements are used to process the same data. One such technique is *dual modular redundancy* (DMR) in which two elements are used. An error is detected when the contents of the two processing elements diverge. Another technique is *triple modular redundancy* (TMR) or N-modular redundancy, which in addition to detecting errors can also locate the faulty element through majority voting. Moreover, the system can continue to execute by masking the faulty element. In such

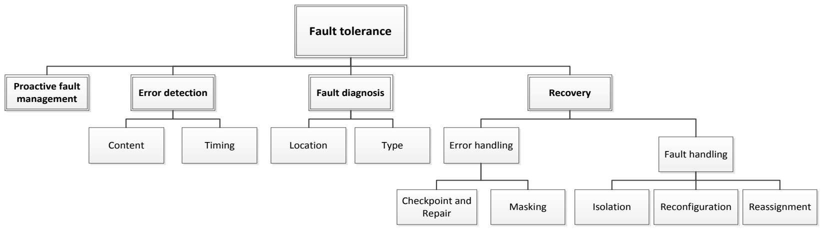

Fig. 2. Classification of steps used for Fault tolerance

systems, the voter also needs to be reliable as it can become a single point of failure.

While N-modular redundancy is used to tolerate hardware faults, N-diversity is used to tolerate software faults, such as, logical bugs left during development. The main idea is that if there is a fault in one version, it can be masked out by using majority voting. Authors in [24] discuss various software fault tolerance techniques using design and data diversity. [7] and [6] are examples of systems which use this technique for tolerating software faults.

For error detection of software running on a single core, fault tolerant systems commonly employ redundant execution at different levels of abstractions, at instruction level [20], process level [22] or virtual machine level [4]. Schemes which work at instruction level have low error detection latencies, while schemes which work at process and virtual machine level allow error to propagate before detecting it. In the absence of faults, these schemes need to make sure that each replica start with the same initial state, executes input data in the same order and perform the same computations. This method is not straightforward to implement, especially for parallel programs running on shared memory multicore/multiprocessor systems. SectionVII discusses the related work done to tackle this problem.

V. Fault diagnosis

Fault diagnosis is the process of identifying **location** and **type** of a fault. Location of a fault can be determined either preemptively, that is, before its activation, or after error has been detected due to its activation, as shown in Figure 1.

Failure identification of a fail-stop component is relatively easy as it stops responding when it fails. Time-out is a common mechanism to detect failures of fail-stop components. For example, in a message passing environment, a permanent failure of a processor would be assumed if it stops sending messages.

In TMR systems, faulty component can be located by a majority voter. Another method to locate faulty components is online self-tests. Through this mechanism, a system can find permanent and sometimes intermittent faults in it by testing itself.

Online self tests can be applied concurrently with application execution and therefore can proactively detect dormant faults by locating failed hardware components. Online self tests can be performed using pure hardware built-in self-test (BIST) [39] approaches or using software based self test (SBST) [26]. The benefit of software based approaches is that they do not require any change to the system hardware. Software based techniques are becoming more relevant with increasing number of cores as a core can be dedicated to perform the self tests on the system.

An example of an online self test scheme is [8] which describes and evaluate three different scheduling policies to find permanent and intermittent faults. In this scheme, the online self test can be performed through either special hardware (BIST) or software (SBST). When a test is performed on a processor, it is logically isolated from the rest of the system, while the task which was being executed by that processor is migrated to another processor for continued operation. In the proposed system, only one task can execute at a time on a processor. Self tests are done periodically and the scheduling policies try to select the idle processors or those which are running low priority tasks for testing. Since the test is performed concurrently with application execution, intermittent faults, such as those that occur during burst of a computing in processor, can also be detected.

Type of a fault can be found by using retry/replay methods. For example, in [25], the same BIST test is applied twice in a row. Knowing that transient faults occur infrequently, it can be assumed that transient fault would not occur twice in a row. Hence, if the test fails both times, the failure is considered permanent. mSWAT [40] can also differentiate between a hardware fault and software bugs for a multithreaded program running on a multicore system. After an error is detected, execution is restarted from the last checkpoint. If no error is detected this time, fault is assumed to be transient or a non-deterministic software bug, otherwise a permanent fault or deterministic software bug is assumed. In that case, execution is replayed on different cores. If the same error occurs again, deterministic software bug is assumed, otherwise permanent fault is assumed. In that case, mSWAT does another replay for further analysis to find the faulty core.

VI. Recovery

When a fault is detected, it is important to recover from it. As shown in Figure 1, in a fault tolerant system, recovery must be done before failure of the system occurs. It can be done by either performing **error handling**, **fault handling** or both. Error handling means to eliminate errors from the system without removing the source of the fault. On the other hand, fault handling is the process of removing the source of fault, to prevent reoccurrence of the fault. Error handling is enough for recovering from transient faults, as it is not necessary to locate the source of the fault for transient faults.

A. Error handling

Two different schemes can be used for error handling, namely **checkpoint and repair** and **masking**. In *checkpoint and repair*, the state of the system is periodically saved (checkpointed) and when an error is detected, it is rollbacked to a previously valid state by using the checkpoint. The benefit of this scheme is that it can be used to tolerate long error detection latencies [12]. On the other hand, in masking, the erroneous components are masked out by using majority voting on the states of redundant components. The state of an erroneous component may be restored by using the state of one of the non-erroneous redundant components [22]. It is a more efficient technique than *checkpoint and repair*, as no rollback is required. However, it is unable to tolerate long error detection latencies. Therefore, introduction of latent faults in such systems needs to be avoided as they may eventually corrupt most of the redundant states to make recovery impossible [36].

Checkpointing can be mainly categorized into coordinated checkpointing and uncoordinated checkpointing. In coordinated checkpointing [30], each process in the system coordinate with each other to take the checkpoint, while in uncoordinated checkpointing, each process separately takes its own checkpoint. The recovery is achieved through a special recovery phase which reforms the global state of the system to perform recovery. Uncoordinated checkpointing is usually avoided however due its proneness to domino effects [37].

Commodity shared memory multicore processors are equipped with memory management unit (MMU) which allows accelerating the checkpointing process by using copy-on-write techniques. It allows incremental checkpointing, that is only saving pages dirtied since the last checkpoint. Authors in [27] and [28] were the first one to implement checkpointing for parallel programs running on shared memory multiprocessor systems by using this technique. Their scheme allows original application to continue execution while checkpointing is performed. This is made possible by giving read only access to the memory pages of the program when starting checkpointing, so whenever something is written to a page for the first time, page fault is trapped by the OS and content of that page saved in the checkpoint besides giving write access back to that page. Authors in [29], improve upon [28] by using translation lookaside buffer (TLB) misses to record data. This avoids the overhead of setting write accesses of pages.

Normally checkpoints are stored in a non-volatile memory due to its reliability. However, schemes like Respec [14] keep the checkpoint as a forked process in linux. This improves efficiency of both storing and restarting from a checkpoint, especially in systems with large amount of RAM. This method is less reliable though. However, its reliability can be increased by using memory scrubbing.

B. Fault handling

Fault handling involves **isolation** of the faulty component and recovering the system from the fault. Moreover, tasks which were being computed on a faulty core need to be reassigned to a working core or a spare core. This is known as **reassignment**. Repair of a faulty component can be done in a reconfigurable system through **reconfiguration** [34]. When hardware resources are exhausted, a reconfigurable system might also emulate a hardware component in software [35], so that system continues to perform albeit with degraded performance.

Isolation of a faulty component is done to make the fault originating from it dormant, so that error is not propagated to the other components in the system. In typical shared memory multicore processors, different processes are run on separate address spaces by using the virtual memory system supported by the MMU. This makes sure that a wild write in one process, due to an uninitialized pointer for example, do not affect the execution of other processes in the system. Therefore the virtual memory provides an efficient scheme for isolation and error confinement. However virtual memory alone would not be not enough to confine errors in case when different processes are communicating through shared memory or at kernel level, since the kernel is itself managing the virtual memory. An error in kernel could bring down the whole operating system.

Hive [31] addresses this issue by using independent kernels known as *cells*. In this way, a fault damages only one cell rather than the whole system. To prevent wild writes from one cell to the memory of another one, each cell uses a *firewall* hardware. On failure of a cell, pages writable from that cell are discarded, which prevents any cell from reading data from those pages. This requires prompt detection of failure of a cell, which Hive does by applying an aggressive fault detection scheme, which includes heuristic checks and a distributed agreement protocol.

Hypervisor based fault tolerance [4] takes a step further by running different guest operating systems in isolated environments. This isolation make sure that failure of one guest OS does not affect the other guest OSs. Moreover, authors in [32] and [33] have exploited the isolation provided by a hypervisor to execute device drivers inside virtual machines for fault tolerance and portability. Due to the isolation provided by the hypervisor, a faulty driver does not impact the rest of the system.

VII. Redundant execution for fault tolerance

Process level and virtual machine level fault tolerant systems apply redundant techniques for fault tolerance. This requires deterministic execution of the redundant components with respect to each other. For this purpose, these systems need to cater for non-deterministic events, such as interrupts, signals, DMAs and non-deterministic functions, such as time of the day. As an example, [13] uses hardware performance counters to count instructions so as to identify the point at which

TABLE I
COMPARISON OF DIFFERENT METHODS FOR DETERMINISTIC REDUNDANT EXECUTION OF SHARED MEMORY MULTITHREADED PROGRAMS

Property / Technique	**Language based (e.g., SHIM)**	**Record / Replay (e.g., Karma)**	**Deterministic execution of programs (e.g., Calvin)**
Scalability	Reasonable	Reasonable	Poor
Programmability	Difficult for arbitrary programs	Easy	Easy
Deadlock prevention	Can be difficult to prevent	Does not prevent	Mutex-based deadlocks can be eliminated

an interrupt occurred in the primary replica and execute the interrupt at the same point of execution in the other replicas.

In multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses and these accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve and therefore an active area of research.

Comparison of the different methods that can be used for deterministic redundant execution is shown in Table I. One way for executing replicas in a deterministic fashion is to use deterministic parallel languages. Examples of such languages are StreamIt [44], SHIM [5] and Deterministic Parallel Java [1]. However, porting programs written in traditional languages to deterministic languages is difficult as learning curve is high for programmers used to programming in traditional languages. Moreover, in languages which are based on the Kahn Process Network Model, such as SHIM, it is difficult to write programs without introducing deadlocks [41].

Deterministic redundant execution at runtime can be done either through hardware, software or a combination of both. Some hardware schemes use record and replay method for achieving deterministic execution. In this method, all interleavings of shared memory accesses by different processors are recorded in a log, which can be replayed to have a replica which follows the original execution. Examples of schemes using this method are Rerun [10] and Karma [42]. These schemes intercept cache coherence protocols to record interprocessor data dependencies, so that they can be replayed later on, in the same order. While Rerun only optimizes recording, Karma optimizes both recording and replaying, thus making it suitable for online fault tolerance. It shows good scalability as well.

Unlike the record/replay method, Calvin [9] executes programs deterministically, that is, given the same input, a program always has the same output. It does so by executing instructions in the form of chunks and later committing them at barrier points. It uses a relaxed memory model, where instructions are committed in such a way that only the total store order (TSO) of the program has to be maintained. An advantage of this method is that mutex-based deadlocks can be eliminated [43]. Moreover, no inter-replica communication is required, thus making this method more dependable than record/replay. The disadvantage of this method though is scalability, as it depends upon barriers to commit chunks.

The disadvantage of existing hardware methods for deterministic execution is that they are applied at system level. They cannot for example, perform deterministic execution of different applications running on a system. Capo [17] is the first scheme to address this issue. It implements a virtualization layer that allows different applications to use the hardware resources for deterministic replay. Non-deterministic events, such as interrupts and signals are handled by the software while for shared memory access interleavings, the underlying hardware for deterministic replay can be used.

Besides hardware methods, software only methods for deterministic execution also exist. One such method is CoreDet [3] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Therefore, it is similar to Calvin, but implemented in software. Since it is implemented in software, it has a very high overhead, 1-11x for 8 cores, as compared to 0.5x-2x for Calvin.

Kendo [19] is a software approach that works only on programs without data races, that is, those that access shared memory only through synchronization objects. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its clock becomes less than those of the other threads. Clock is calculated from retired stores, is paused when waiting for a lock and resumed after lock is acquired. Since this method requires global communication among threads for reading clock values, it also has limited scalability.

Respec [14] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rollbacks and re-execute from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed, which can induce a large overhead.

VIII. CONCLUSION

In this paper we discussed related work done on online fault tolerance techniques with focus on techniques for shared memory multicore/multiprocessor systems. We have discussed steps which are taken to achieve fault tolerance, which are *proactive fault management*, *error detection*, *fault diagnosis* and *recovery*. Proactive fault management is a precautionary step to prevent failures of components in the system, whereas error detection is performed to detect errors before they lead to failure of the system. We also discussed fault diagnosis techniques which are used to locate failed components and to check the type of a fault. Moreover, we discussed recovery techniques such as checkpoint and repair, reconfiguration and reassignment. Finally we discussed related work to perform deterministic redundant execution of parallel programs running on shared memory multicore/multiprocessor systems, which is still an active area of research.

ACKNOWLEDGMENT

This research is supported by Artemis through the SMECY project (grant 100230).

REFERENCES

[1] http://dpj.cs.uiuc.edu.

[2] R. C. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, vol. 1, pages 17 –22, March 2001.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, vol. 38, pages 53–64, March 2010.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[5] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 264–272, New York, NY, USA, 2005. ACM.

[6] B. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association.

[7] R. Kapitza H. P. Reiser, F. J. Hauck and W. Schrder-Preikschat. Kemari: Vm synchronization for fault tolerance. In *European Dependable Computing Conference*, pages 67–68, 2006.

[8] O. Heron, J. Guilhemsang, N. Ventroux, and A. Giulieri. Analysis of on-line self-testing policies for real-time embedded multiprocessors in dsm technologies. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 49 –55, 2010.

[9] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333 –334, February 2011.

[10] D. R. Hower and M .D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 265 –276, 2008.

[11] Y. Huang, C. Kintala, N. Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *IN PROCEEDINGS OF IEEE INTL. SYMPOSIUM ON FAULT TOLERANT COMPUTING, FTCS 25*, 1995.

[12] W W. Hwu and Y. N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *Computers, IEEE Transactions on*, vol. 36, pages 1496 –1514, December 1987.

[13] C .M. Jeffery and R .J .O. Figueiredo. Towards byzantine fault tolerance in many-core computing platforms. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 256 –259, 2007.

[14] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. *SIGPLAN Not.*, vol. 45, pages 77–90, March 2010.

[15] A. Mahmood and E .J. McCluskey. Watchdog processors: Error coverage and overhead. *Dig. 15th Annu. Int. Symp. Fault-Tolerant Comput., FTCS-15, Ann Arbor, MI.*

[16] A. Mahmood and E .J. McCluskey. Concurrent error detection using watchdog processors-a survey. *Computers, IEEE Transactions on*, vol. 37, pages 160 –174, 1988.

[17] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. *SIGPLAN Not.*, vol. 44, pages 73–84, March 2009.

[18] S. Nomura, M. D. Sinclair, C. Ho, V. Govindaraju, M. Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. *SIGARCH Comput. Archit. News*, vol. 39, pages 201–212, June 2011.

[19] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, vol. 44, pages 97–108, March 2009.

[20] G .A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243 – 254, 2005.

[21] A. M. Saleh, J. J. Serrano, and J. H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *Reliability, IEEE Transactions on*, vol. 39, pages 114 –122, April 1990.

[22] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, pages 135 –148, 2009.

[23] G. Vallee, C. Engelmann, A. Tikotekar, T. Naughton, K. Charoenpornwattana, C. Leangsuksun, and S. L. Scott. A framework for proactive fault tolerance. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 659 –664, 2008.

[24] Z. Xie, H. Sun, and K. Saluja. A survey of software fault tolerance techniques.

[25] A. Sanyal, S. Alam, and S. Kundu. Bist to detect and characterize transient and parametric failures. *Design Test of Computers, IEEE*, vol. 27, pages 50 –59, 2010.

[26] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. Software-based self-testing of embedded processors. *IEEE Trans. Comput.*, vol. 54, pages 461–475, April 2005.

[27] K. Li, J. F. Naughton, and J. S. Plank. Real-time concurrent checkpoint for parallel programs. *SIGPLAN Not.*, vol. 25, pages 79–88, February 1990.

[28] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, pages 874–879, August 1994.

[29] J. Liao and Y. Ishikawa. A new concurrent checkpoint mechanism for real-time and interactive processes. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 47 –52, 2010.

[30] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, vol. 34, pages 375–408, September 2002.

[31] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[32] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, vol. 6, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[33] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[34] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. *SIGARCH Comput. Archit. News*, vol. 35, pages 470–481, June 2007.

[35] A. Miele A software framework for dynamic self-repair in embedded socs exploiting reconfigurable devices. *International Conference on Automation, Quality and Testing, Robotics*, vol. 2, pages 1–6, 2010.

[36] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pages 448–465, May 2006.

[37] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[38] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, pages 11 – 33, 2004.

[39] H. Al-Asaad, B. T. Murray, and J. P. Hayes. Online bist for embedded systems. *IEEE Des. Test*, vol. 15, pages 17–24, October 1998.

[40] S. K. S. Hari, M. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: Low-cost hardware fault detection and diagnosis for multicore systems. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 122 –132, December 2009.

[41] B. Jiang, E. Deprettere, and B. Kienhuis. Hierarchical run time deadlock detection in process networks. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 239 –244, October 2008.

[42] A. Basu, J. Bobba, and M D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.

[43] E. D. Berger T. Liu, C. Curtsinger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*, October 2011.

[44] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.

3 FAULT TOLERANCE USING RECORD/REPLAY

SUMMARY
The abundant computational resources available in multicore systems have made it feasible to implement otherwise prohibitively intensive tasks on consumer grade systems. However, these systems integrate billions of transistors to implement multiple cores on a single die, thus raising reliability concerns, as smaller transistors are more susceptible to transient as well as permanent faults.

A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the state machine replication approach. In this approach the replicated copies of a process (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions (such as *gettimeofday*) deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the original and replica processes perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

In this chapter, we describe a software based efficient fault tolerance scheme that is implemented using a user-level library and does not require a modified kernel. The record and replay of synchronization operations is made efficient and scalable by eliminating atomic operations and true and false sharing of cache lines. Moreover, the error detection mechanism is optimized to perform memory comparisons of the replicas efficiently in user-space. With our initial algorithm for record/replay, the overhead is up to 46% for 4 threads, which is reduced to maximum 18% with an improved algorithm. This is lower than comparable systems published in literature.

This chapter is based on the following papers.

1. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *A user-level library for fault tolerance on shared memory multicore systems*, Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on, pp. 266-269, 18-20 April 2012

2. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Efficient software-based fault tolerance approach on multicore platforms*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, pp. 921-926, 18-22 March 2013

A User-level Library for Fault Tolerance on Shared Memory Multicore Systems

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

***Abstract*—The ever decreasing transistor size has made it possible to integrate multiple cores on a single die. On the downside, this has introduced reliability concerns as smaller transistors are more prone to both transient and permanent faults. However, the abundant extra processing resources of a multicore system can be exploited to provide fault tolerance by using redundant execution. We have designed a library for multicore processing, that can make a multithreaded user-level application fault tolerant by simple modifications to the code. It uses the abundant cores found in the system to perform redundant execution for error detection. Besides that, it also allows recovery through checkpoint/rollback. Our library is portable since it does not depend on any special hardware. Furthermore, the overhead (up to 46% for 4 threads), our library adds to the original application, is less than other existing approaches, such as *Respec*.**

I. Introduction

Although the shrinking transistor size has made it possible to implement multiple cores on a single die, it has also made reliability a concern, as smaller transistors are more prone to both transient [1] as well as permanent [2] faults. However, the abundant processing resources of a multicore system can be exploited to provide fault tolerance through redundant execution.

One way to use the abundant processing resources to provide fault tolerance is by using the state machine replication approach [3]. For multithreaded programs running on shared memory multicore systems, it is required that threads of the replicas access shared memory in the same order. In other words, shared memory accesses should be deterministic. Our library ensures this. Overall it provides the following features.

- Efficient deterministic execution in presence of lock-based shared memory accesses.
- Optimized memory comparison of the replicas for error detection.
- Checkpoint/rollback to perform recovery from transient errors.

In Section II we discuss the background and related work. In Section III, we discuss the overview of our library, while in Section IV, we discuss its implementation. In Section V, we present and discuss our results. We finally conclude the paper with Section VI.

II. Background and related work

Fault tolerance is achieved by three major steps, error detection, isolation and recovery [4]. With redundant execution, an error is detected if the replicas diverge. Since any kind of divergence is used to detect an error, it is important to remove any source of divergence which is not due to an error. The only sources of non-determinism in a single threaded program are non-deterministic functions, such as *gettimeofday* and asynchronous signals, while for a multithreaded program, there is also non-determinism due to shared memory accesses. Moreover, shared memory accesses are usually much more frequent as compared to non-deterministic functions and asynchronous signals, which makes implementing efficient state machine based replication more difficult for a multithreaded program, running on a shared memory multicore system, than for a single threaded application.

Two main approaches, record/replay and deterministic multithreading, exist for this purpose. In record/replay, the order of shared memory accesses on the original processes are recorded so that they can be replayed by the other replicas. On the other hand, with deterministic multithreading, given the same input, a process always performs the same ordering of shared memory accesses. It has to be noted though that for non-deterministic functions, such as *gettimeofday* and asynchronous signals, record/replay is the only viable method. Our library uses the record/replay approach.

For record/replay, both hardware and software-based methods exist. Karma [5] and DeLorean [6] are examples of hardware-based approaches. While Karma intercepts the cache coherence protocols to record inter-processor data dependencies and later use these recorded data dependencies to replay, DeLorean uses a relaxed memory model, where each processor executes instructions in chunks concurrently and an arbiter is used to commit the chunks. Replaying is done by replaying the chunks in the order in which they were committed. Respec [7] is a software-based method. It logs the ordering of acquisition and release of synchronization objects, such as mutexes, to make replicas acquire the synchronization objects in the same order. It also performs checkpoint/rollback to perform recovery.

For deterministic multithreading, also, both hardware and software-based methods exist. An example of hardware-based approach is Calvin [8]. It executes a program deterministically by executing instructions in the form of chunks and committing them at barriers points deterministically. Kendo [9] is a software based approach that only deals with programs

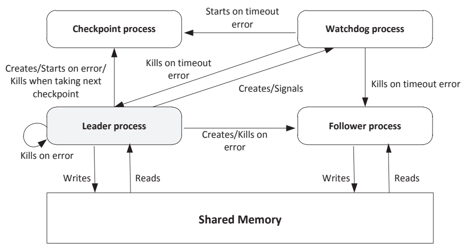

Fig. 1. Data flow diagram of our fault tolerance scheme

without data races. For efficient deterministic execution, it performs load balancing by only allowing a thread to acquire a synchronization object if that thread has executed less instructions than the other threads. The number of instructions are calculated by counting the retired stores.

III. OVERVIEW OF THE LIBRARY

Our library is intended to reduce probability of failures in the presence of transient faults. The data flow diagram of our fault tolerance scheme is shown in Figure 1. Initially, the leader process creates its replica (follower process) and the watchdog. The execution is divided into time slices (epochs). At the end of each epoch, the memories of the leader and follower processes are compared by the leader. If no divergence is found, a checkpoint is taken and output to files or screen is committed. The previous checkpoint is also deleted. The checkpoint is basically a suspended process which is identical to the leader process at the time the checkpoint is taken. If a divergence is found at the end of an epoch, the leader process signals the checkpoint process to start and kills itself and its follower. When the checkpoint process starts, it becomes the leader and creates its own follower. It might also happen that the leader or follower process are unable to reach the end of an epoch, due to some error which hangs them. In that case, the watchdog process detects those hangs by using timeouts and signals the checkpoint process to start. The watchdog process itself is less vulnerable to transient faults as it remains idle most of the time.

IV. IMPLEMENTATION

This section discusses the implementation of our library. In Section IV-A, we discuss how the follower process is created. In Section IV-B, we discuss our memory allocation technique, which is followed by Section IV-C, where we explain how we are able to deterministically access the shared memory through mutexes. Section IV-D discusses our error detection mechanism, while Section IV-E discusses recovery. Lastly, in Section IV-F, we discuss how our library handles I/O and non-deterministic functions such as *gettimeofday*. Note that currently our library does not support deterministic replaying of asynchronous signals, which is left as future work.

A. Follower creation

Our library assumes that threads in the application are created once at the start of the application. Therefore, we create the follower process at point in the code where the threads are created. For this purpose, we replace the *pthread_create* function with *r_pthread_create*, which internally calls *pthread_create* function and indirectly calls the Linux's *fork* function.

To make sure that the follower threads have the same stack contents as the leader, our library itself allocates the memory used for thread stacks. These allocated stacks are then passed as attributes to the *pthread_create* function (called from *r_pthread_create*). When the leader calls the *fork* function, although all threads, beside the one calling it, die in the forked process, the stack contents are still present in the memory. Therefore, to recreate a thread with the same stack in the follower, we call *pthread_create* with the same stack attribute. For thread identification, we use a thread local variable, so that we can relate a thread in the follower process with that in the leader process.

For making sure that a follower thread also has the same register values as the corresponding leader thread, the thread's start routine passed as an argument to the *pthread_create* function (called from *r_pthread_create*), is not the start routine itself but wrapper *thread_start* that then calls the start routine, which is provided as a parameter to it. For the leader process, this function calls our barrier function *r_pthread_barrier_wait* in the beginning. When *r_pthread_barrier_wait* is called for the first time, each thread of the leader saves its registers, including the instruction pointer, by using the C *setjmp* function. The follower is itself created by the main thread by calling *fork* function, from within the *r_pthread_barrier_wait* function. The newly forked process then recreates the threads and each thread jumps to the same location as the thread in the leader process by using the C *longjmp* function. This is done by having the forked process also pass the *thread_start* function as start routine to *pthread_create* function. But unlike the leader process, the follower threads call *longjmp* at the beginning of the *thread_start* function.

Note that since the main thread is replicated by the *fork* process, we do not need to recreate it. However, *r_pthread_barrier_wait* needs to be inserted just after the code where the threads are created, so that all the threads are at a barrier point when the follower is forked.

B. Memory allocation

In an operating system with Address Sapce Layout Randomization (ASLR), malloc can be non-deterministic. This is because *malloc* internally uses *mmap* for allocating memory blocks of large sizes and *mmap* can be non-deterministic. Therefore, whenever the memory allocator uses *mmap*, we make sure the follower has the same address returned for *mmap* by calling *mmap* with MAP_FIXED flag and the address returned by the leader process.

The variables used by our library (not related to original program execution) to perform deterministic execution, may have different values for the leader and follower processes, for example, the flag used to distinguish the leader process from the follower process. For these variables, we use a separate

```
function R_PTHREAD_MUTEX_LOCK(ref pthread_mutex_log_t m)
    if $isLeader$ then
        $lock(m.mutex)$
        $m.leaderClock = m.leaderClock + 1$
        while $m.threadClock[tid] > 0$
        end while
        $m.threadClock[tid] = m.leaderClock$
    else
        while not $(m.threadClock[tid] == (m.followerClock + 1))$
        end while
        $m.threadClock[tid] = 0$
        $lock(m.mutex)$
    end if
end function

function R_PTHREAD_MUTEX_UNLOCK(ref pthread_mutex_log_t m)
    if not $isLeader$ then
        $m.followerClock = m.followerClock + 1$
    end if
    $unlock(m.mutex)$
end function
```

Fig. 2. Pseudocode for deterministic lock and unlock

memory, which is allocated with *mmap*. This memory is not compared for error detection.

C. Deterministic shared memory accesses

For redundant deterministic execution, it is necessary that the leader and follower processes perform shared memory accesses in the same order. Since we assume that a program has no data races and all synchronization operations are done using mutexes, we provide functions *r_pthread_mutex_lock* and *r_pthread_mutex_unlock* for deterministically locking and unlocking a mutex. A mutex is enclosed in a special data structure, known as *pthread_mutex_log_t*, which also contains a pointer to clocks for that mutex to aid in deterministic execution. The memory region to hold the mutex clocks is shared between the leader and follower processes.

Our deterministic locking and unlocking algorithms for mutexes are shown as Figure 2. The benefit of this scheme is that it uses less memory as compared to schemes that use producer/consumer queues, such as Respec [7]. Secondly, by wrapping the pointer to the mutex clocks in the same data structure as the mutex, we avoid the overhead of using a hash table, which is used by Respec. Lastly, our algorithm is written such that it exploits the strict memory consistency model of multicore x86 (memory ordering respects transitive visibility and stores to the same location have a total order) and thus avoid using atomic variables (which incur significant overhead due to use of memory fences) on such systems.

D. Error detection

At regular intervals (epochs) of one second, the leader and follower processes calculate checksums by performing modular sums of the contents of the dirtied (modified) memory pages, which are then compared by the leader. If a discrepancy is found, a fault is detected. Follower keeps its checksum in the shared memory so that the leader can read it from there for comparison. We perform memory comparison at barriers which are already found in the program. If insufficient number of barriers are found in the program, the programmer can insert our library function *r_potential_barrier_wait* in

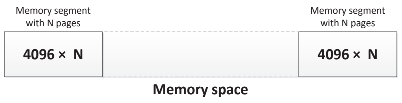

Fig. 3. Memory pages can be grouped into segments to reduce the overhead of memory comparison for error detection

the code. This function will create a barrier (by calling *r_pthread_barrier_wait*) only if the program has reached the end of an epoch.

To note down dirtied pages, at start of each epoch, we give only read access to memory pages, so that a page faults can be trapped to note down dirtied pages. To reduce the number of such page faults however, we exploit the concept of spatial locality of data and segmented memory into multiple pages, as shown in Figure 3. A write on any part of a read protected segment of N pages is handled by giving write access to all the N pages in that segment. This improves the execution considerably, as discussed in Section V, where we discuss the performance evaluation.

The watchdog process is used to detect hangs and recover from them. At the end of each epoch, the leader process sends a signal to the watchdog process to signal that it is not hung.

E. Recovery

For fault recovery, we use checkpoint/rollback. Checkpointing is done by forking a process and suspending it. If the leader process detects an error or the Watchdog detects a hang, a signal is sent to the checkpoint process to start execution. The leader and follower processes are also killed. The checkpoint process now becomes the new leader and forks its own follower. The checkpoint process also resets the mutex clocks (which exist in shared memory), since they could have been corrupted by an error.

Creation of the checkpoint process is very similar to creation of the follower (see Section IV-A), with the difference being that the checkpoint process is suspended in the beginning. Only when it is signalled to start, it recreates the threads and starts execution.

F. I/O and non-deterministic functions

For I/O, our library allows deterministic I/O for sequential file access and screen write. Write to a file or screen is only performed after making sure that no error occurred during an epoch. For that purpose, no output is committed during an epoch. Instead it is buffered. Therefore, our library defines a structure *r_FILE*, which not only contains the FILE pointer, but also the buffer. The *r_fopen* function returns a pointer to this structure. The buffers are then committed at the end of an epoch after comparing the buffer contents of the leader and follower by using checksums. For sequential file reading, we provide functions *r_fread* and *r_fscanf*. The *r_FILE* structure also contains the file offset value at the last epoch, so that the file pointer can be rewinded to the previous value in case of rollback.

For non-deterministic functions such as *gettimeofday*, our library allows the programmer to create a deterministic wrapper

TABLE I
PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmarks	**Threads**	**Epochs**	**Locks**	**Pages compared**	**Original time (ms)**	**Deterministic exec time (ms) & Overhead**	**Overall time (ms) & Ovheread**
Fluidanimate	2	3	4313983	33890	1988	2298 (16%)	2379 (20%)
	4	2	7466285	25380	1205	1696 (41%)	1712 (42%)
Ocean	2	4	5046	104504	2432	2459 (1%)	3091 (27%)
	4	3	10092	80618	1890	1895 (0%)	2119 (12%)
Water-nsq	2	3	125047	12624	1837	1859 (1%)	1861 (1%)
	4	2	188142	24912	1090	1112 (2%)	1170 (7%)
Radiosity	2	2	6124778	6920	920	1140 (24%)	1153 (25%)
	4	1	6170016	12616	589	854 (44%)	865 (46%)
Raytrace	2	1	121958	2344	1116	1216 (9%)	1260 (13%)
	4	1	121960	2348	663	700 (6%)	738 (11%)

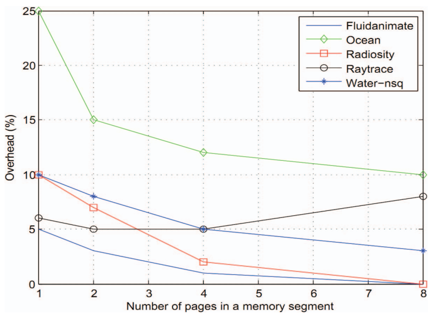

Fig. 4. Reduction in overhead by grouping memory into segments

by using functions *r_log_data* and *r_read_data*. *r_log_data* is called by the leader process to log the outputs of that function, while the follower process reads the outputs by calling the *r_read_data* function.

V. PERFORMANCE EVALUATION

We selected 5 benchmarks, one from the PARSEC [10] and four from the SPLASH-2 [11] benchmark sets. We ran all our benchmarks on an 8 core (dual socket with 2 quad cores), 2.67 GHz Intel Xeon processor with 32GB of RAM, running CentOS Linux version 5, with kernel 2.6.18. We used gcc 4.4.4 and optimization level -O3 to compile our results. The results are shown in Table I. For each benchmark, we show the results for 2 and 4 threads (For redundant execution, 4 threads means 8 threads in total). We compare the original execution time with deterministic execution time (excluding overhead of error detection, checkpointing and watchdog) and the overall time which includes all the overheads. We used memory segments of size 4 (See Section IV-D for discussion on memory grouping). For *fluidanimate*, which has high lock frequency our library only adds an overhead of 42%, while Respec adds an overhead of 67%. Also for *ocean*, which has high memory consumption, Respec adds an overhead of 43%, while our library adds an overhead of just 12% due to the optimized memory comparison scheme.

Figure 4 shows the impact of grouping memory pages on performance. We show the results for memory segment sizes of 1, 2, 4 and 8. Note that the overhead shown is after subtracting the overhead of deterministic execution. We can see that for an applications like *Ocean* which has high memory usage, we get significant performance gains using page grouping. However, grouping too many pages can also cause the application to compare more pages which have not been actually modified by that application, thus creating unnecessary overhead. This is evident for *Raytrace* which has lower memory usage than other benchmarks. However, for 4 pages, all five benchmark show performance gain.

VI. CONCLUSION

In this paper, we described the design and implementation of a user-level library for fault tolerance of multithreaded user-level applications running on shared memory multicore systems. Our library requires programmer to make little modifications to the program for providing fault tolerance. It allows creation of a multithreaded redundant process for detecting errors and provides facility of checkpointing and rollback for recovery. We also applied several optimizations to speedup the execution, like reducing memory for logging, which is required for record/replay, and optimizing memory comparison for error detection. Empirical measurements on tested benchmarks show that the overhead does not exceeds 46% for four threads.

ACKNOWLEDGMENT

This research is supported by Artemis through the SMECY project (grant 100230).

REFERENCES

[1] R. Baumann, “Soft errors in advanced semiconductor devices-part i: the three radiation sources,” *Device and Materials Reliability, IEEE Transactions on*, vol. 1, no. 1, pp. 17 –22, mar 2001.

[2] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, “Sampling + dmr: practical and low-overhead permanent fault detection,” in *Proceeding of the 38th annual international symposium on Computer architecture*, ser. ISCA ’11, 2011, pp. 201–212.

[3] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: a tutorial,” *ACM Comput. Surv.*, vol. 22, pp. 299–319, December 1990.

[4] H. Mushtaq, Z. Al-Ars, and K. Bertels, “Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems,” in *Design and Test Workshop (IDT), 2011 IEEE 6th International*, dec. 2011, pp. 12 –17.

[5] A. Basu, J. Bobba, and M. D. Hill, “Karma: scalable deterministic record-replay,” in *Proceedings of the international conference on Supercomputing*, ser. ICS ’11, 2011, pp. 359–368.

[6] P. Montesinos, L. Ceze, and J. Torrellas, “Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA ’08, 2008, pp. 289–300.

[7] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, “Respec: efficient online multiprocessor replayvia speculation and external determinism,” in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS ’10, 2010, pp. 77–90.

[8] D. Hower, P. Dudnik, M. Hill, and D. Wood, “Calvin: Deterministic or not? free will to choose,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 333 –334.

[9] M. Olszewski, J. Ansel, and S. Amarasinghe, “Kendo: efficient deterministic multithreading in software,” *SIGPLAN Not.*, vol. 44, pp. 97–108, March 2009.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT ’08, 2008, pp. 72–81.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: characterization and methodological considerations,” *SIGARCH Comput. Archit. News*, vol. 23, pp. 24–36, May 1995.

Efficient Software-Based Fault Tolerance Approach on Multicore Platforms

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

Abstract—**This paper describes a low overhead software-based fault tolerance approach for shared memory multicore systems. The scheme is implemented at user-space level and requires almost no changes to the original application. Redundant multithreaded processes are used to detect soft errors and recover from them. Our scheme makes sure that the execution of the redundant processes is identical even in the presence of non-determinism due to shared memory accesses. It provides a very low overhead mechanism to achieve this. Moreover it implements a fast error detection and recovery mechanism. The overhead incurred by our approach ranges from 0% to 18% for selected benchmarks. This is lower than comparable systems published in literature.**

I. Introduction

The abundant computational resources available in multicore systems have made it feasible to implement otherwise prohibitively intensive tasks on consumer grade systems. However, these systems integrate billions of transistors to implement multiple cores on a single die, thus raising reliability concerns, as smaller transistors are more susceptible to both transient [12] as well as permanent [13] faults.

A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the state machine replication approach [14]. In this approach the replicated copies of a process (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions (such as *gettimeofday*) deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the original and replica processes perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

Different software-based solutions have been proposed, for deterministic execution of shared memory multithreaded programs on multicore processors, such as DTHREADS [9] and CoreDet [1], which are too slow to be used for practical purposes. On the other hand, Kendo [7], while an efficient solution, suffers from portability problem as it requires the use of deterministic hardware performance counters, which are not available on many platforms [10]. Respec [5] is a record/replay approach for fault tolerance, that requires kernel modification and also does not have highly efficient method of memory comparison for error detection. Moreover, it does not perform deterministic execution very efficiently for benchmarks with high lock frequencies.

In this paper, we describe a software based efficient fault tolerance scheme that performs the following.

1) The scheme is implemented using a user-level library and does not require a modified kernel.
2) Record and Replay of synchronization operations is made efficient and scalable by eliminating atomic operations and true and false sharing of cache lines.
3) The error detection mechanism is optimized to perform memory comparisons of the replicas efficiently in user-space.

In Section II we discuss the background and related work, while in Section III, we discuss the implementation. In Section IV, we evaluate the performance of our scheme. We finally conclude the paper with Section V.

II. Background and related work

For error detection of software running on a single core, fault tolerant systems commonly employ redundant execution at different levels of abstraction, at instruction, process or virtual machine level [15]. Schemes which work at instruction level have low error detection latencies, especially those which operate at hardware level. On the other hand, schemes which work at process and virtual machine level allow error to propagate before detecting it. Another important issue in process level and virtual machine level systems is that they need to cater for non-deterministic events, such as interrupts and non-deterministic functions, such as time of the day. They need to make sure that execution of the replicas is deterministic with respect to each other. Such schemes usually use the concept of primary and backup replicas, where the primary is responsible for logging information about the non-deterministic events to be used by the backups. For this purpose, non-deterministic events such asynchronous signals have to be executed at the same point in the code by the replicas. As an example, [16] defers asynchronous signal handling to known points in the code, such as function calls, system calls or backward branches.

In multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses. These accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve.

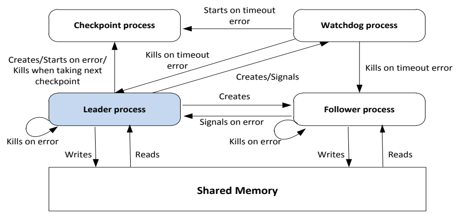

Fig. 1. Data flow diagram of our fault tolerance scheme

Lately, effort has been done to create deterministic languages, that ensure deterministic execution of a program. Examples of programming languages designed for deterministic parallel execution are StreamIt [8] and SHIM [3]. However porting programs written in traditional languages to deterministic languages is difficult as the learning curve is high for programmers used to programming in traditional languages. Therefore, deterministic execution at runtime is still the only viable solution to most users.

One such method for runtime deterministic execution is CoreDet [1] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Since this method requires bulk syncrhonous quantas, it has a very high overhead (1-11x for 8 cores) and limited scalability.

Kendo [7] is a software approach that works only on programs without data races, that is, those that access shared memory only through synchronization objects. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its logical clock, which is used to perform deterministic execution, becomes less than those of the other threads. Since this method requires global communication among threads for reading clock values, it also has limited scalability.

Respec [5] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rollbacks and re-execute from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed. It uses producer-consumer queues. A queue is shared between a thread in the leader and its corresponding thread in the follower and is used to record logical clocks for mutexes. Each recorded operation atomically increments a clock. Since having a producer-consumer queue for each mutex will require a large memory, *Respec* only uses fixed number of clocks, that is, 512. The hash of the address of a mutex is used to point to its logical clock. A thread in the follower process only acquires a mutex when its logical clock matches that recorded by the corresponding thread of the leader.

III. FAULT TOLERANCE SCHEME

Our fault tolerant scheme is intended to reduce probability of failures in the presence of transient faults. The data flow diagram of our fault tolerance scheme is shown in Figure 1. Initially, the leader process (which is the original process highlighted in the figure) creates the watchdog and follower processes. The follower process is identical to the leader process and follows the same execution path. The execution is divided into time slices known as epochs. An epoch starts and ends at a program barrier. At the end of each epoch, the memories of the leader and follower processes are compared by the follower. If no divergence is found, a checkpoint is taken and output to files or screen is committed. The previous checkpoint is also deleted. The checkpoint is basically a suspended process which is identical to the leader process at the time the checkpoint is taken. If a divergence is found at the end of an epoch, the follower process signals an error to the leader process which in turn signals the checkpoint process to start and kills itself and its follower. This can also happen inside an epoch, if the follower sees that the parameters (of synchronization functions or system calls) logged by the leader do not match those read by the follower. When the checkpoint process starts, it becomes the leader and creates its own follower. It might also happen that the leader or follower processes are unable to reach the end of an epoch, due to some error which hangs them. In that case, the watchdog process detects those hangs by using timeouts and signals the checkpoint process to start. The watchdog process itself is less vulnerable to transient faults as it remains idle most of the time.

At this moment, our fault tolerant scheme does not work with programs that use inter process communication (such as through pipes and shared memory). The only form of I/O allowed is disk I/O and screen output. Moreover, our scheme assumes that there are no data races in the program. Lastly, we have not added functionality to handle asynchronous signals. However, this functionality can be added for user space by handling asynchronous signals at synchronous points, such as system calls, as done by Scribe [17].

In Section III-A, we discuss how we allow deterministic execution of the replicas. This is followed by Section III-B which discusses error detection. Finally in Section III-C, we discuss our recovery mechanism.

A. Deterministic execution

For deterministic execution, we need to ensure that replicas use the same memory addresses. We also need to ensure determinism in the presence of non-deterministic functions and shared memory accesses. Moreover, we need to make sure that the leader and follower processes use the same memory addresses. For this we need to have a deterministic memory allocation scheme. Finally we also need to make sure that we have deterministic I/O. Below we discuss how we handle these issues.

1) Replica creation: Our library creates a follower from the leader process by using *fork* system call, at the beginning and also when a rollback is done. This is because at a rollback, the checkpoint process becomes the leader and creates its own follower, which uses the same memory addresses as the leader process. We use our own version of *pthread_create* function to make sure that the leader and follower processes use the same stack addresses for the threads. For this purpose, the leader process logs these addresses to be consumed by the follower. For thread identification, we use a thread local variable, so that we can relate a thread in the follower process with that in the leader process.

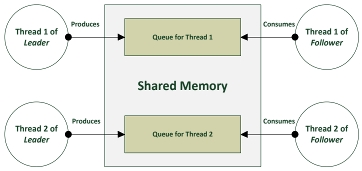

Fig. 2. Communication between the leader and follower processes for deterministic execution

Algorithm 1 Pseudocode for deterministic lock

```
function R_PTHREAD_MUTEX_LOCK(ref pthread_mutex_log_t m)
    q = GetQueue(tid)                                   ▷ There is a separate queue for each thread
    if isLeader then
        r = lock(m.mutex)
        if r == 0 then                                  ▷ Only if lock call is successful, increment the clock
            m.clock = m.clock + 1                       ▷ m.clock does not need to be atomic
        end if
        while !pushq(q, MUTEXLOCK, m.mutex, m.clock, r)
        end while
        return r
    else                                                ▷ Follower
        while not !popq(q, ref type, ref mutex, ref clock, ref r)
        end while
        if type != MUTEXLOCK and mutex != m.mutex then  ▷ Logged parameters do not match
            SignalErrorAndExit()
        end if
        if r != 0 then
            return r
        end if
        while (m.clock+1) != clock
        end while
        lock(m.mutex)
        m.clock = m.clock + 1
        return 0
    end if
end function

function PUSHQ(q, type, addr, clock, r)                 ▷ Called by Leader
    lindex = GetLeaderQIndex(tid)
    if checkQElementsForZero(lindex) then
        q[lindex].type = type
        q[lindex].addr = addr
        q[lindex].clock = clock
        q[lindex].r = r + 1
        SetLeaderQIndex((lindex + 1) %QCAPACITY)
        return TRUE
    else
        return FALSE
    end if
end function

function POPQ(q, ref type, ref addr, ref clock, ref r)  ▷ Called by Follower
    findex = GetFollowerQIndex(tid)
    if checkQElementsForNonZero(qindex) then
        type = q[findex].type
        addr = q[findex].addr
        clock = q[findex].clock
        r = q[findex].r - 1
        setQElementsToZero(findex)
        SetFollowerQIndex((findex + 1) %QCAPACITY)
        return TRUE
    else
        return FALSE
    end if
end function
```

2) Memory allocation: We implement our own memory allocation functions to allocate memory deterministically. In an operating system with Address Space Layout Randomization (ASLR), malloc can be non-deterministic. This is because *malloc* internally uses *mmap* for allocating memory blocks of large sizes and *mmap* can be non-deterministic. Therefore, whenever the memory allocator uses *mmap*, we make sure the follower has the same address returned for *mmap* by calling *mmap* with MAP_FIXED flag and the address returned by the leader process.We also make sure that threads of the leader and follower processes call the *malloc* function in the same order by internally using a mutex, which is locked and unlocked deterministically.

The variables used by our library (not related to original program execution) to perform deterministic execution, may have different values for the leader and follower processes, for example, the flag used to distinguish the leader process from the follower process. For these variables, we use a separate memory, which is allocated with *mmap*. This memory is not compared for error detection.

3) Deterministic shared memory accesses: For redundant deterministic execution, it is necessary that the leader and follower processes perform shared memory accesses in the same order. For this purpose, a mutex is enclosed in a special data structure, which also contains a pointer to clocks for that mutex to aid in deterministic execution. Whenever a thread in the leader process acquires a mutex, it increments the mutex's clock. A thread in the follower only acquires the same mutex in its execution, when its clock matches that for the corresponding thread in the leader.

We create our own deterministic versions of pthread's synchronization functions, such as *pthread_mutex_lock*, *pthread_mutex_unlock*, *pthread_trylock*, *pthread_cond_wait*, *pthread_broadcast*, *pthread_barrier_wait* etc. Since *pthread_mutex_lock* is the mostly used and is also used in our implementation of other pthread synchronization functions, we discuss our *pthread_mutex_lock* algorithm here, which is shown in Algorithm 1. We also have our own versions of data structures for representing the synchronization objects, for example, *pthread_mutex_log_t* instead of *pthread_mutex_t*. Here *m* represents an object of *pthread_mutex_log_t* structure which holds a mutex and its clocks. There is one such object for each mutex in the program. Therefore, deterministic access to a mutex is independent of other mutexes in the program, hence improving scalability.

When a leader thread acquires a mutex, it increments the leader's clock for that mutex and also records that value in a circular queue, so that the follower can acquire the thread when its clock reaches one less than the same value. The communication between the leader and follower processes is shown in Figure 2. After acquiring the mutex, the follower also increments its clock for that mutex. Unlike Respec which uses a hash table of 512 entries to keep clocks for all the syncrhonization objects, we use a separate clock for each mutex. The benefit of this is that we can avoid using atomic variables for accessing the clocks, as clock can be incremented after acquiring the lock.

We also optimize the queue access by avoid using atomic variables and avoiding true and false sharing of cache lines. For that purpose, we use a lockless queue as shown by *pushq* and *popq* functions in Algorithm 1. This is unlike *Respec* which uses atomic operations if necessary to access the queue. The typical method of using a lockless queue (which we call *naive* in this paper) is to use shared tail and head indexes. Since in this method, producer and consumer read the head or tail indexes at the same time when the other is writing to it, this causes cache trashing. Hence it is a *true sharing* problem. We avoid this by having local indexes for producer (leader) and consumer (follower). The check for emptiness and fullness is done by checking the data value instead. Producer only writes to the queue when all the fields of the queue element it is about to write to, are zero, while the consumer only reads when all the fields of the queue element are non-zero. Here, since the value of *r*, which represents the result returned by a synchronization function can be zero, We add one to its value while pushing and subtract one from it when popping. We make sure that the indexes for leader and follower do not share the same cache line by having sufficient padding between them. This makes sure that we do not have the problem of *false sharing*.

4) System Calls, Non-deterministic functions and I/O: We use LD_PRELOAD to preload the system call wrappers found in *glibc* with our own version which perform logging. This is possible, because most of the system calls can be and are usually called through their user-space wrapper functions. This method will not work however, if for example, a system call is made without using the wrapper function, for example, by using inline assembly. So, with our library, the programmer needs to make sure to not make a system call directly. Since the glibc library sometimes also make system calls directly, for example, by making the clone system call in *pthread_create* function, we provide our version of *pthread_create*. We also provide our own version of non-deterministic functions such as *gettimeofday* and *rand* and preload them using LD_PRELOAD. The leader performs logging of the parameters and output of these non-deterministic functions and system calls. The logged parameters are used by the follower to check for errors (by checking for discrepancies), whereas the logged output is just read by the follower. Furthermore, each non-deterministic function and system call is protected by a deterministic lock so that the leader and follower processes perform these calls in the same order.

For I/O, our library allows deterministic I/O for sequential file access and screen write. Write to a file or screen is only performed after making sure that no error occurred during an epoch. For that purpose, no output is committed during an epoch. Instead it is buffered. Our library overrides the *write* and *read* system call wrappers to allow buffering of the data. The buffers are committed at the end of an epoch after comparing the buffer contents of the leader and follower by using hash-sums. For this purpose, each file opened for writing is allocated a special buffer. It is important that addresses of these buffers are the same for the leader and follower process. For this purpose, we use a deterministic memory allocation scheme like the one described in Section III-A2. For sequential file reading, the file offset value is saved at the end of each epoch, so that the file can be rewinded to the previous value in case of rollback.

B. Error detection

At regular intervals of 1 second, known as epochs, dirtied memories of the leader and follower processes are compared. However, the epoch time is reduced to 100 ms if a file or screen output occurs during the epoch. Instead of comparing each memory one by one, the leader and follower processes calculate hash-sums of the dirtied (modified) memory pages, which are then compared. If a discrepancy is found, a fault is detected. The hash-sums are calculated much faster by using the CRC32 instruction of the SSE4.2 instruction set found on modern x86 processors.

The comparison is made even faster by assigning each thread to calculate hash-sum of different portions of the memory. Follower keeps its hash-sums in shared memory so that the leader can read it from there for comparison. We perform memory comparison at barriers which are already found in the program rather than stopping and creating a barrier. This improves the performance, as threads already wait for each other at barriers. If insufficient barriers are found in the program, the programmer can insert calls to function *potential_barrier_wait*, which is provided by our library. This function creates a barrier only when required, that is at the end of an epoch.

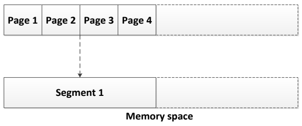

Fig. 3. Memory pages can be grouped into segments to reduce the overhead of memory comparison for error detection

Since our scheme runs at the user-space level, we cannot note down dirtied pages while handling page faults (from the kernel), the way *Respec* does, which is the most efficient method possible. We take special steps to improve its performance at user-space level.

At start of each epoch, we give only read access to allocated memory pages. Whenever a page is accessed for writing, the OS sends a signal to the accessing thread. In the signal handler, the address of the memory page is noted down and both read and write accesses are given to that memory page. In this way, we only need to compare the dirtied memory pages at the end of an epoch. Sending signals on each memory page access violation can slow down execution. Therefore, to reduce the number of such signals, we exploit the concept of spatial locality of data and segmented memory into multiple pages, as shown in Figure 3. A write on any part of a read protected segment of N pages is handled by giving write access to all the N pages in that segment. This improves the execution considerably, as discussed in Section IV, where we discuss the performance evaluation.

Some functions, like that for comparing memories, change the stacks differently for the leader and follower threads. For those purposes, we switch to a temporary stack, so that the original stack remains unaltered from such functions.

The watchdog process is used to detect hangs and recover from them. At the end of each epoch, the leader process sends a signal to the watchdog process to signal that it is not hung. In that signal, the process ID of the checkpoint is also sent, so that the watchdog is able to start the checkpoint process in case it detects hang of leader or follower process. Hangs are detected by using timeout. Besides sending the process ID of the checkpoint, the leader also sends process ID of itself and the follower process when it forks the follower, so that the watchdog process can kill the leader and follower processes before starting the checkpoint process.

C. Recovery

As discussed previously, for fault recovery, we use checkpoint/rollback. whenever the leader takes a checkpoint, it kills the previous checkpoint. If the leader process detects an error, or the Watchdog process detects a hang, a signal is sent to the last checkpoint process, so that the checkpoint process can start execution. The leader and its follower are killed at that point. The checkpoint process then assumes the role of the leader and forks its own follower. It also creates a new checkpoint. Moreover, it resets the mutex clocks (which exist in shared memory), since they could have been corrupted by an error. Checkpoints are taken only at barrier points. For creating a multithreaded follower, we have implemented a special *multithreaded fork* function that replicates the leader process to create the follower.

IV. Performance evaluation

We selected 8 benchmarks, two from the PARSEC [2] and six from the SPLASH-2 [11] benchmark sets. We ran all our benchmarks on an 8 core (dual socket with 2 quad cores), 2.67 GHz Intel Xeon processor with 32GB of RAM. All programs were compiled using gcc 4.4.4 with optimization level -O3. The results are shown in Table I.

For each benchmark, we show the results when the benchmark runs for 2 and 4 threads. Number of epochs executed are shown in the third column, while number of synchronization operations performed by each process is shown in the fourth one. This is followed by the number of barriers and total number of memory pages compared for error detection. Then the table shows the redundant execution time, which is the time to execute two instances of the same application. For redundant execution, each instance is executed on one of the two different quad core processors of the dual socket system. Next we show the time for deterministic execution scheme, which is execution without performing error detection and checkpointing but only deterministic locking and unlocking of the mutexes. This is followed by the overall execution (with error detection, checkpointing and Watchdog process). Next we show the overheads of the deterministic and overall execution with respect to the redundant time. For overall execution, the results are shown with memory grouping size of 4.

A. Results

Figure 4 shows the improvement that we get by avoiding atomic variables and having an optimized queue. The left bars are obtained by running the benchmarks with 2 threads while right bars are obtained with 4 threads. The lower portion of the bar shows the overhead with our lockless queue and our method for keeping the clocks for mutexes, while the middle portion shows the additional overhead that we get when we use Respec's method of using a Hash Table for mutex clocks. The upper most portion shows the additional overhead by using naive lockless queue.

We can see that for *fluidanimate*, which has a high lock frequency, we have a significant improvement in performance of deterministic execution. Furthermore, our method of using separate clocks for each mutex is more scalable than Respec's method of using limited clocks and accessing them through a Hash table, that requires atomic operations. The scalability here can be assessed by the fact that for two threads, our scheme and Respec's scheme perform similarly, while for four threads, our scheme performs far better. From this result, we can predict that our scheme will have even better results compared to Respec for larger number of cores. Furthermore, our lockless queue also shows much better scalability than a

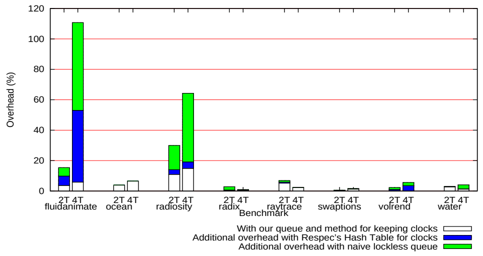

Fig. 4. Deterministic execution overhead

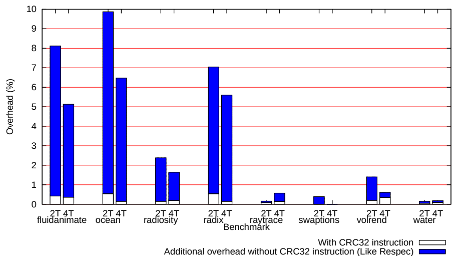

Fig. 5. Memory comparison overhead with and without CRC32 instruction

naive lockless queue due to avoiding true and false sharing of cache lines. Note that we do not get as much improvement for *radiosity*, which also has a high lock frequency, because it uses much fewer mutexes than *fluidanimate*, and hence fewer clocks, causing more contention in communication between the leader and follower threads.

Figure 5 shows the improvement that we get from using the CRC32 instruction (as opposed to Respec which does not use that instruction) for calculating hash sums for error detection. The results are especially impressive for benchmarks which modify higher number of pages, such as *fluidanimate*, *ocean* and *radix*.

In Figure 6, we show the overall results in the form of bar graphs, with each factor shown separately. Due to the optimizations we discussed and reduction in epoch overhead, which will be discussed in the next section, the overhead never exceeds 18% for four threads and is negligible for benchmarks with small memory usage and low lock frequencies.

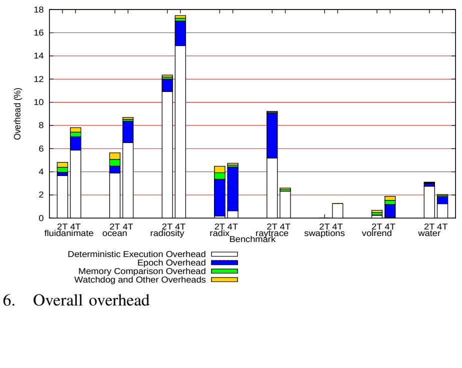

Fig. 6. Overall overhead

TABLE I. PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmarks	Threads	Epochs	Synch Ops	Barriers	Pages modified	Redundant exec time (ms)	Deterministic exec time (ms)	Overall time (ms)	Det exec overhead w.r.t Redt exec time(%)	Overall overhead w.r.t Redt exec time (%)
fluidanimate	2	3	8689200	161	25238	2073	2149	2168	4%	5%
	4	2	16909680	161	25136	1294	1370	1392	6%	8%
ocean	2	3	10092	10763	103398	2819	2929	3008	4%	7%
	4	2	20184	10763	74518	1840	1960	2029	7%	10%
radiosity	2	2	8981938	19	10378	1199	1330	1347	11%	12%
	4	2	8900495	19	10344	887	1019	1043	15%	18%
radix	2	2	18	12	57706	1072	1074	1138	0%	6%
	4	1	54	12	35170	624	628	666	1%	7%
raytrace	2	2	243914	2	200	1214	1277	1325	5%	9%
	4	1	243918	2	148	690	706	702	2%	2%
swaptions	2	1	77020	1	6	510	510	510	0%	0%
	4	1	77020	1	8	239	242	242	1%	1%
volrend	2	3	459760	241	160	2490	2496	2503	0%	1%
	4	2	463922	241	154	1443	1444	1462	0%	1%
water	2	2	125047	664	474	1889	1941	1948	3%	3%
	4	2	188142	664	550	1125	1139	1148	1%	2%

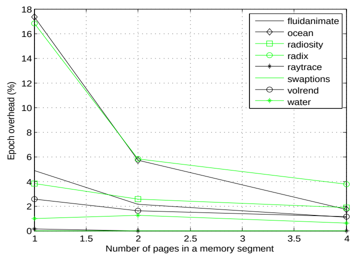

Fig. 7. Reduction in overhead by grouping memory into segments

B. *Impact of Grouping Memory Pages*

Figure 7 shows the impact of grouping memory pages (see Section III-B) on the performance. The overhead shown is the *epoch* overhead which mainly consists of the overhead of signals for noting down dirtied memory pages. We can see that for applications like *ocean* and *radix* which have high memory usage, we get significant performance gains using page grouping. However, it has to be noted that there is a limit to the number of pages which can be grouped for optimal performance, as grouping too many pages will cause the application to compare more pages which have not been actually modified by that application, thus creating unnecessary overhead.

V. CONCLUSION

In this paper, we described the design and implementation of a user-space level leader/follower based fault tolerance scheme for multithreaded applications running on multicore processors. We applied several optimizations to speedup the execution, like avoiding atomic variables, true and false sharing of cache lines for recording/replaying synchronization operations, reducing signals sent by the OS on page faults (used to note down dirtied pages) and using the CRC32 instruction from SSE4.2 instruction set to greatly improve error checking performance. Empirical measurements on tested benchmarks show that the overhead does not exceeds 18% for four threads. We compared our results with Respec and showed that our scheme is more efficient in performing deterministic execution and comparing memories for error detection. Moreover, we showed that by grouping memory pages, we considerably reduced the overhead of signals used for noting modified memory pages.

ACKNOWLEDGMENT

This research has been funded by the projects Smecy 100230, iFEST 100203 and REFLECT 248976.

REFERENCES

[1] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. PACT '08, pages 72–81.

[3] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. EMSOFT '05, pages 264–272.

[4] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. HPCA '11, pages 333 –334, feb.

[5] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, ASPLOS '10, pages 77–90.

[6] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. ISCA '08, pages 289–300.

[7] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44:97–108, March 2009.

[8] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196.

[9] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. SOSP '11, pages 327–336.

[10] V. Weaver and S. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141 –150.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995.

[12] R. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, 1(1):17 –22, mar 2001.

[13] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. ISCA '11, pages 201–212.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[15] H. Mushtaq, Z. Al-Ars, and K. Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. IDT 2011, pages 12–17.

[16] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *Dependable and Secure Computing, IEEE Transactions on*, 6(2):135 –148, 2009.

[17] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):155–166, June 2010.

4

DETERMINISTIC MULTITHREADING

SUMMARY
In this chapter, we present *DetLock*, a runtime system to ensure deterministic execution of multithreaded programs running on multicore systems. DetLock does not rely on any hardware support or kernel modification to ensure determinism. For tracking the progress of the threads, logical clocks are used. Unlike previous approaches, which rely on non-portable hardware to update the logical clocks, Detlock employs a compiler pass to insert code for updating these clocks, thus increasing portability. Moreover, unlike the state of the art approaches, that update the logical clocks only after execution, DetLock can update the logical clocks ahead of time, thus improving the performance of deterministic multithreading further. For 4 cores, the average overhead of these clocks on tested benchmarks is brought down from 16% to 2% by applying several optimizations. Moreover, the average overall overhead, including deterministic execution, is 14%. We also employed DetLock for fault tolerance.

This chapter is based on the following papers.

1. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *DetLock: Portable and Efficient Deterministic Execution for Shared Memory Multicore Systems*, High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 721-730, 10-16 Nov. 2012

2. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Efficient and highly portable deterministic multithreading (DetLock)*, Computing, vol. 96, pp. 1131-1147, 2014

3. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Fault tolerance on multicore processors using deterministic multithreading*, Design and Test Symposium (IDT), 2013 8th International, 16-18 Dec. 2013

2012 SC Companion: High Performance Computing, Networking Storage and Analysis

DetLock: Portable and Efficient Deterministic Execution for Shared Memory Multicore Systems

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{*H.Mushtaq, Z.Al-Ars, K.L.M.Bertels*}*@tudelft.nl*

***Abstract*—Multicore systems are not only hard to program but also hard to test, debug and maintain. This is because the traditional way of accessing shared memory in multithreaded applications is to use lock-based synchronization, which is inherently non-deterministic and can cause a multithreaded application to have many different possible execution paths for the same input. This problem can be avoided however by forcing a multithreaded application to have the same lock acquisition order for the same input.**

In this paper, we present *DetLock*, which is able to run multithreaded programs deterministically without relying on any hardware support or kernel modification. The logical clocks used for performing deterministic execution are inserted by the compiler. For 4 cores, the average overhead of these clocks on tested benchmarks is brought down from 20% to 8% by applying several optimizations. Moreover, the overall overhead, including deterministic execution, is comparable to state of the art systems such as *Kendo*, even surpassing it for some applications, while providing more portability.

I. INTRODUCTION

Single threaded programs are much easier to test, debug and maintain than their multithreaded counterparts. This is because the only source of non-determinism in them are interrupts or signals, which are rare. On the other hand, multithreaded programs have a frequent source of non-determinism in the form of shared memory accesses. Due to this, multithreaded programs suffer from repeatability problems, which means that running the same program with the same input can result different outputs. This repeatability problem makes multithreaded programs hard to test and debug. Furthermore, it is also difficult to build fault tolerant versions of these programs. This is because fault tolerance systems usually depend upon replicas (identical copies of redundant processes) to detect errors.

If access to shared data is not protected by synchronization objects in a multithreaded program, we can have race conditions, which may produce unexpected results. Running a program with race conditions deterministically does not avoid the problem of having unexpected results with those race conditions, but just makes sure that we get the same output with the same input.

The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [2], which has a maximum overhead of 11x for 8 cores. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [11]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. Two such approaches are Calvin [4] and DMP [14]. They use the same concept as *CoreDet* for deterministic execution but make use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, we can relax the requirements to improve efficiency. For example, Kendo [9] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without race conditions. The authors of *Kendo* call it *Weak Determinism*. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [8], *Weak Determinism* is sufficient for most well written multithreaded programs. Therefore, *DetLock* also only supports *Weak Determinism*.

The basic idea of *Kendo* is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, *Kendo* still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counter that is deterministic [12]. Secondly, *Kendo* needs modification of the kernel to allow reading from the hardware performance counters for deterministic execution.

To overcome portability issues faced by *Kendo*, our tool *DetLock* has a completely software-based approach of updating the logical clocks. The code for updating the clocks is inserted through an LLVM [5] compiler pass. Since, LLVM is a popular open source compiler framework available on many platforms, our approach is portable across a wide range of platforms. Moreover, it requires no modification of the kernel. We can sum up the contribution of this paper as follows.

- A portable mechanism to update logical clocks for *Weak Deterministic* execution that depends upon the compiler rather than using hardware performance counters, since many platforms have no such deterministic counters available.
- A User-space approach to update the logical clocks that does not require modifying the kernel.
- A number of optimization steps to reduce the overhead of the code used to update the logical clock and improve the performance of deterministic execution.

This paper is organized as follows. In Section II, we discuss the background and related work, while in Section III, we give an overview of DetLock's architecture. This is followed by Section IV where we present the optimization methods used to improve the performance of *DetLock*. In Section V, we evaluate the performance of our scheme. We finally conclude the paper with Section VI.

II. Background and Related Work

Single threaded programs are mostly deterministic in behavior. We say mostly because interrupts and signals can introduce non-determinism even in single threaded programs. However, these non-deterministic events are rare. On the other hand, in multithreaded programs running on multicore processors, shared memory accesses are a frequent source of non-determinism.

One way to ensure determinism of multithreaded programs is to write code for them in a deterministic parallel language. Examples of such languages are StreamIt [10], SHIM [3] and Deterministic Parallel Java [1]. The disadvantage of this approach is that porting programs written in traditional languages to deterministic languages is difficult as the learning curve is high for programmers used to programming in traditional languages. Moreover, in languages which are based on the Kahn Process Network Model, such as SHIM, it is difficult to write programs without introducing deadlocks [7].

Deterministic execution at runtime can be done either through hardware or software. Calvin [4] is a hardware approach that executes instructions in the form of chunks and later commits them at barrier points. It uses a relaxed memory model, where instructions are committed in such a way that only the total store order (TSO) of the program has

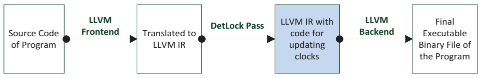
Source Code of Program
LLVM Frontend
Translated to LLVM IR
DetLock Pass
LLVM IR with code for updating clocks
LLVM Backend
Final Executable Binary File of the Program

Figure 1: *DetLock* modifies the LLVM IR code by inserting code for updating logical clocks

to be maintained. DMP [14] uses a similar relaxed memory approach. The disadvantage of hardware approaches is that they are restricted to the platforms they were developed for.

Besides hardware methods, software only methods for deterministic execution also exist. One such method is CoreDet [2] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Therefore, it is similar to Calvin, but implemented in software. Logical clocks are used for deterministic execution. Since CoreDet is implemented in software, it has a very high overhead, possibly upto 11x for 8 cores, as compared to the maximum 2x for Calvin. Another similar approach is DTHREADS [11]. It runs threads as separate processes, so that memories which are modified can be tracked down through the memory management unit. Only at synchronization points such as locks, barriers and thread creation for example, it updates the shared memory from the local memories of the threads. Therefore, it avoids the overhead of using bulk synchronous quantas like CoreDet and also does not have the need to maintain logical clocks like CoreDet. However, the overhead for programs with high lock frequency or large memory usage is still very high.

Since performing deterministic execution in software alone is inefficient, Kendo [9] relaxes the requirements by only working for programs without race conditions (*Weak Determinism*). It does not use any hardware besides deterministic hardware performance counters found in some processors. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its clock becomes less than those of the other threads, with ties broken with thread IDs. Clock is calculated from retired stores, is paused when waiting for a lock and resumed after the lock is acquired. Kendo still suffers from portability problems as it requires hardware performance counters which are deterministic. Many platforms, including many x86 platforms, do not have any deterministic hardware performance counter [12]. Moreover, *Kendo* requires modification of the kernel to read from such hardware performance counters.

One technique related to deterministic multithreading is record/replay. In this method, all interleaving of shared memory accesses by different cores/processors are recorded in a log, which can be replayed to have a replica which follows the original execution. Examples of schemes using this method are Rerun [16] and Karma [15]. These schemes intercept cache coherence protocols to record inter-processor data dependencies, so that they can be replayed later on, in the same order. While Rerun only optimizes recording, Karma optimizes both recording and replaying, thus making it suitable for online fault tolerance. It shows good scalability as well. The disadvantage of record/replay approaches as compared to deterministic multithreading is that they require a large memory for recording. Moreover, when used for fault tolerance, the redundant processes need to communicate with each other, as one replica records the log while the other reads from it.

Respec [6] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rolls-back and re-executes from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed, which can induce a large overhead.

III. Overview of the Architecture

In this section, we discuss the architecture of *DetLock* and the application programming interface (API) that it provides to the programmer.

A. Architecture

We use Kendo's algorithm to perform deterministic execution. However, unlike *Kendo* which requires deterministic hardware performance counters, which are not available on many platforms, we insert code to update logical clocks at compile time. This also means we do not need to modify the kernel which is required by *Kendo* to read from performance counters. Figure 1 shows the point of compilation where the *DetLock* pass executes, which is between the point where the LLVM IR (Intermediate Representation) code is translated to the final binary code by the LLVM backend.

The unit of our logical clock is one instruction. For instructions which take more than one clock cycles, the logical clock is updated according to the approximate number of clock cycles they take. However, to keep our discussion simple, in this paper, for *DetLock* one instruction equals one logical clock count.

The Kendo's method of acquiring locks deterministically is illustrated in Figure 2. In this figure, an example is given for a process with two threads. If Thread 1 is trying to acquire a lock when its logical clock is 1029, it will not be able to do so if Thread 2's clock is at 329, because of being less than 1029. But, as soon as Thread 2's clock get past 1029, Thread 1 will acquire the lock.

So basically our purpose is not only to reduce the code that updates the clocks but also to update the clocks as soon as possible. In fact, at compile time it is possible to increment the clock even before instructions are executed. For example, if we know that a leaf function (a function with no function calls) executes fixed amount of instructions, we can increment the logical clock before executing any instruction of that function. So for example, if Thread 2 in Figure 2 has logical clock of 329 and is about to execute

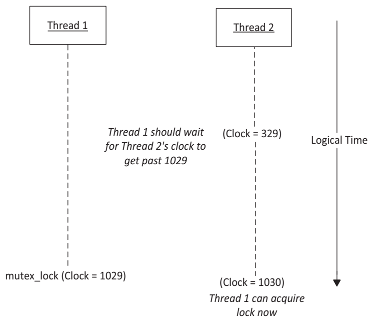
Thread 1
Thread 2
Thread 1 should wait for Thread 2's clock to get past 1029
(Clock = 329)
Logical Time
mutex_lock (Clock = 1029)
(Clock = 1030)
Thread 1 can acquire lock now

Figure 2: Kendo's method of acquiring locks for deterministic execution

a leaf function with 701 instructions, we can add 701 right away to its logical clock, making it 1030 from 329. In this way, Thread 1, whose clock is at 1029, can acquire the lock without waiting for Thread 2 to actually have executed that amount of instructions.

Therefore in all optimizations we apply, besides trying to reduce the clock update overhead, we also try to increment the clock as soon as possible. Without any optimization, we update the clock at start of each of the basic block of LLVM IR. If there is a function call inside that block, we split that block, such that each block either contains no function call or starts and end with a function call. Then we update the clock at the top of each block if that block contains no function calls, otherwise we update the clocks in between the function calls. By splitting blocks in such a way, we can more easily apply optimizations.

To illustrate the effect of our optimizations, we are going to show how the optimizations change the example function shown in Figure 3. This function is taken from the *Radiosity* benchmark of SPLASH 2 [13]. The clocks associated with each block are shown at the right of the assignment operators. A block in parallelogram shape implies that it contains one or more function calls.

B. Application Programming Interface

We provide our own functions for locks, barriers and thread creation for deterministic execution. They internally use the *pthread* library. However, it is not necessary for the programmer to modify the code to use them. A header file is provided by us that replaces the definition of these functions with ours. The header file can be specified in the makefile, thus making it unnecessary to modify source code files. Moreover, the code to initialize the clock for the main thread is inserted by the compiler.

It has to be noted that since our method depends upon the compiler to insert clocks for deterministic execution, it is not possible to increment the clocks in functions which are implemented in a library (Since they have not been compiled

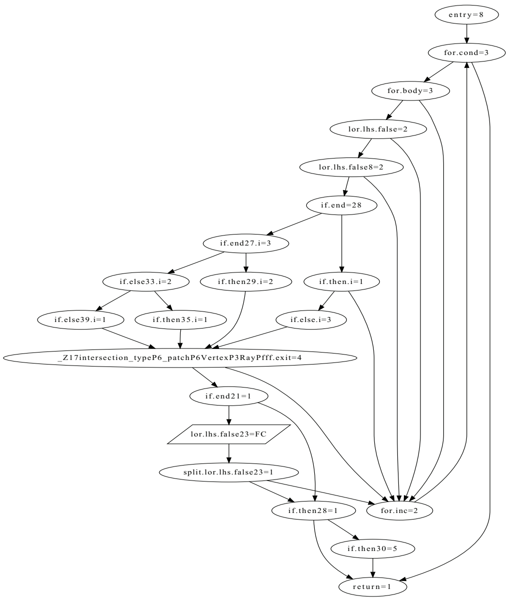

Figure 3: Example function for discussing the optimizations

```
function isClockable(out Int avg, ref Function f)
    if hasLoops(f) or hasUnclockedFunctions(f) then
        return false
    end if
    clocks = getClocksOfAllPaths(f)
    avg = mean(clocks)
    s = std(clocks)
    r = range(clocks)
    if r > (m / 2.5) or s > (m / 5) then
        return false
    end if
    return true
end function

function updateClockableFuncList
    modified = true
    while modfied do
        modified = false
        for all f in Program do
            if (not clockableList.find(f)) and isClockable(avg, f) then
                removeClockFromFunction(f)
                clockableList.insert(f, avg)
                modified = true
            end if
        end for
    end while
end function
```

Figure 4: Pseudocode for Optimization 1 (Function Clocking)

with our pass). This problem also exists for functions which are built in the compiler, as LLVM generates no code for them at IR level. For many built-in functions such as *memset* and math functions, we just keep an estimate of the instructions they take and increment the clock accordingly. For *memset* and other functions which depend upon the size parameter, we increment the clock considering the size parameter. Since most built-in functions are simple, we can use an estimate for them. We provide a text file (instructions estimate file) for such purpose, where these functions can be defined with the approximate number of instructions they take along with their dependency on input parameters. However, this is not always possible for functions in shared libraries. One way is to ignore them and the other way is to add them in the *instructions estimate file* if possible (If the instructions count for them can be approximated satisfactorily).

Another concern are functions which internally use locks, such as *malloc*. For such functions, we provide our own implementation which replaces the locks with our own deterministic locks.

IV. Performance Optimizations

We apply several optimizations to reduce the clock updating overhead. Moreover, we try to increment clocks as soon as possible so that waiting time for threads who are waiting for other thread's clocks to go past them is reduced. Clock updating code is removed from the blocks whose clocks are made zero by our optimizations. In this paper, we highlight such blocks with gray color. The optimizations are discussed below.

A. *Optimization 1 (Function Clocking)*

As discussed in Section III-A, the sooner the clocks are updated, the better, and leaf functions with only one basic block are perfect candidates for such an optimization. Clocks can be removed from such functions and instead be added to the basic blocks calling such functions. Besides functions with only one blocks, our method also considers leaf functions with multiple blocks, given that there are no loops in such functions. If our pass sees that all possible paths taken by such a function do not differ by much, we calculate the mean value for all possible paths and use that mean value to update the clock. The criteria we have set is that the minimum and maximum clock difference of all possible paths should not be more than the mean value divided by 2.5. Moreover the standard deviation between all the different paths should not be greater than one fifth of the mean value. This is checked by calling the *isClockable* function shown in Figure 4.

We call such leaf functions as clocked functions. By intuition, we can judge that it is also possible to clock functions which call only clocked functions. In this way, we can even clock functions which are not necessarily leaf functions. The *UpdateClockableFunctList* function shown in Figure 4 shows how we do this. Our algorithm greedily searches for all such functions. This is done by first checking

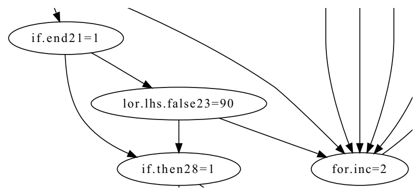

Figure 5: Part of example function after applying Optimization 1 (Function Clocking)

for all the functions in the program to see if they can be clocked and making them clocked functions if possible. If at the end, we see that one or more functions were added to the clocked functions list, which is signaled by the *modified* flag, we iterate over all the functions once again to search for more clockable functions. We keep on repeating this process until no more function is added to clockable functions' list in an iteration.

Part of example function after applying this optimization is shown in Figure 5. Originally, the block **lor.lhs.false23** had a function call at the start, therefore it was split in such a way that **lor.lhs.false23** contained the function call and **split.lor.lhs.false23** the remaining instructions in that block. However, this optimization notices that the function called in **lor.lhs.false23** is clockable, thus no splitting of the block is done and the mean number of instructions from all paths of that function are added to the clock of **lor.lhs.false23**. Moreover, clocks from all the blocks of the called function are removed.

B. Optimization 2 (For Conditional Blocks)

This optimization deals with if-else and switch statements and consists of two parts, **a** and **b**. The part **a** is a precise optimization, meaning that no estimation of clocks is used. They are just rearranged, so as to remove clocks from blocks if possible and incrementing the clock as soon as possible. On the other hand, part **b** is not necessarily precise, but we make sure that the clock does not diverge significantly after that pass.

1) Part a: This optimization is based on the principle that if a block has two or more successors, we can make the successor with the least clock zero and subtract its original value from all its siblings, while also adding its original clock to the parent block. Another principle of this optimization is that if all predecessors of a merge block have that merge block as their only successor, the clocks could be shifted from the merge node to them. The pseudocode of this optimization is shown in Figure 6. The *meetsOpt2aCondNodeRequirements* call on line 7 checks if a node meets the first principle, while *meetsOpt2aMergeNodeRequirements* call on line 15 checks for the second principle. Note that for the first principle, *meetsOpt2aCondNodeRequirements* also makes

```
 1: function UPDATEOPT2ACLOCKS(ref bool modified, ref BasicBlock bb)
        ▷ When this function is called Entry block of a function is passed as bb
 2:     if visitedList.find(bb) then
 3:         return
 4:     end if
 5:     visited.insert(bb)
 6:     modified = false
 7:     if meetsOpt2aCondNodeRequirements(bb) then
 8:         if allSuccessorsHaveNonZeroClock(bb) then
 9:             modified = true
10:         end if
11:         min = minimumOfSucessors(bb)
12:         setClock(bb, GetClock(bb) + min)
13:         subtractFromAllSuccessors(bb, min)
14:     else
15:         if meetsOpt2aMergeNodeRequirements(bb) then
16:             pushClockUp(bb)
17:         end if
18:     end if
19:     succList = getAllSuccessors(bb)
20:     for all succ in succList do
21:         updateOpt2aClocks(modified, succ)
22:     end for
23: end function

24: function PUSHCLOCKUP(ref BasicBlock mergeBlock)
25:     clock = getClock(mergeBlock)
26:     removeClock(mergeBlock)
27:     predList = getAllPredecessors(mergeBlock)
28:     for all pred in predList do
29:         setClock(pred, GetClock(pred) + clock)
30:         if meetsOpt2aMergeNodeReq(pred) then
31:             pushClockUp(pred)
32:         end if
33:     end for
34: end function

35: function APPLYOPT2A
36:     for all f in Program do
37:         modified = true
38:         while modfied do
39:             visitedList.clear()
40:             updateOpt2aClocks(modified, f.entry())
41:         end while
42:     end for
43: end function
```

Figure 6: Pseudocode for Optimization 2a

sure that no unclocked function call exists in the parent block and its successors. Moreover, it makes sure that the parent block is dominating the successors, that is, the successors are not merge blocks. Similarly *meetsOpt2aMergeNodeRequirements* also makes sure that none of the blocks in consideration have unclocked function calls. It also makes sure that the merge block is not a loop header.

It should be noted that after having parsed all the blocks of a function and applying this optimization, if it is still possible to apply this optimization once more to reduce clock updating code, it is applied. This is done by checking the *modified* flag.

The example function after applying one pass of this optimization is shown in Figure 7. The sequence of events that will happen are given below (Refer to Figure 3 for original clock values).

- **if.then.i** is made 0 by **if.end**, which itself becomes 29

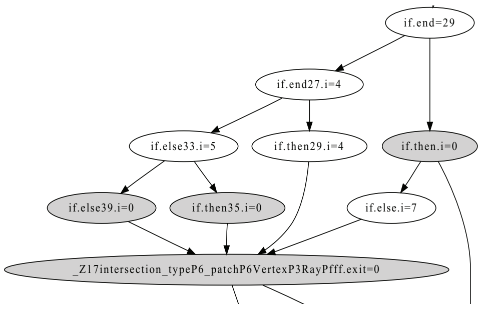

Figure 7: Part of example function after applying first iteration of Optimization 2a

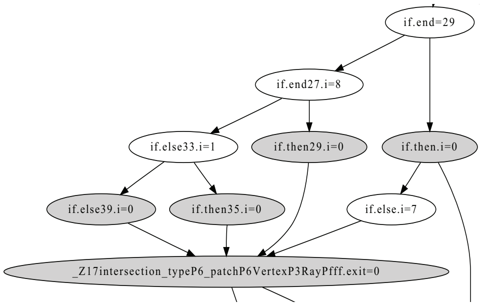

Figure 8: Part of example function after second and final iteration of Optimization 2a

and makes **if.end27** equal to 2.

- **if.then.i** reaches the merge node **_Z17intersection_typeP6_patchP6...** through **if.else.i**.
- Merge node **_Z17intersection_typeP6_patchP6...** becomes 0 while propagating its clock to all of its 4 predecessors, which are **if.else39**, **if.then35.i**, **if.then29.i** and **if.else.i**, whose values now become 5, 5, 6 and 7 respectively.
- **if.end27** subtracts 2 from **if.else33** and **if.then29.i** to make them 0 and 4 respectively while itself becoming 4.
- **if.else33.i** takes value of 5 from **if.else39.i** and **if.then35.i** after making them 0.

Note that after applying this one pass, further optimization is still possible, but after the second pass (shown by Figure 8), no further optimization is possible.

2) Part b: The part **b** of this optimization deals with if conditions, such as those made by the blocks **if.end21**, **lor.lhs.false23** and **if.then28** in Figure 10. The pseudocode for this optimization is shown in Figure 9. The variable *swSucc* in Figure 9 represents the block in the middle, which is **lor.lhs.false23** in this example, while *endSucc* represents the merge node, which is **if.then28** for this example. The *meetsOpt2bRequirements* function call at line 6 checks if a

```
 1: function UPDATEOPT2BCLOCKS(ref BasicBlock bb)    ▷ When this
    function is called, Entry block of a function is passed as bb
 2:     if visitedList.find(bb) then
 3:         return
 4:     end if
 5:     visited.insert(bb)
 6:     swSucc, endSucc, meetsReq = meetsOpt2bRequirements(bb)
 7:     if meetsReq then
 8:         modifyClocks(bb, swSucc, endSucc)
 9:         updateOpt2bClocks(endSucc)
10:         swSuccList = getAllSuccessors(swSucc)
11:         for all succ in swSuccList do
12:             if succ != endSucc then
13:                 updateOpt2bClocks(endSucc)
14:             end if
15:         end for
16:     else
17:         succList = getAllSuccessors(bb)
18:         for all succ in succList do
19:             updateOpt2bClocks(succ)
20:         end for
21:     end if
22: end function

23: function APPLYOPT2B
24:     for all f in Program do
25:         visitedList.clear()
26:         updateOpt2bClocks(f.entry())
27:     end for
28: end function
```

Figure 9: Pseudocode for Optimization 2b

pattern like the one shown in Figure 10 is formed.

If the block **lor.lhs.false23** was not jumping to **for.inc**, that is, it had no successor other than **if.then28**, we could have straight away removed clock updating code from **if.end21** and added its clock value to **if.then28** to make it 2. That optimization, like part **a** would have been precise. However, since **lor.lhs.false23** has one more successor, our algorithm checks to see how much clock divergence we will get by removing clock from **if.end21**. The criteria we keep is that if the divergence is less than one tenth, we proceed with the optimization. In this case, by removing clock from **if.end21**, if we jumped to **for.inc** from **lor.lhs.false23**, the divergence would be 1/93, which is well below one tenth. Therefore, we proceed with it. The example function now becomes what is shown in Figure 10.

Note that this pass also determines if clock has to be removed from the upper block (**if.end21** in this case) or the lower block (**if.then28** in this case). We prefer to remove it from the lower block (and add it to upper block) so that clock is incremented ahead of time. However, in certain cases, we remove it from the upper block (and add it to lower block). One such case is when the upper block is at a higher loop depth than the lower block. Removing clock from upper clock is beneficial here since it is in a more critical path and therefore we save clock updating overhead. Another case where we remove clock from the upper block (and add to the lower block) is when the lower block has a higher clock than the upper block and middle block has

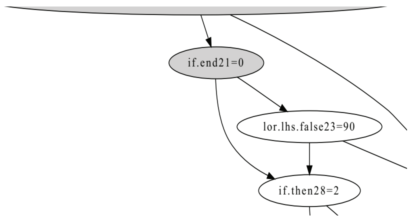

Figure 10: Part of example function after applying Optimization 2b

```
1: function UPDATEOPT3CLOCKS(ref BasicBlock bb)    ▷ When this function is called, Entry block of a function is passed as bb
2:     if visitedList.find(bb) then
3:         return
4:     end if
5:     visited.insert(bb)
6:     if meetsOpt3Requirements(bb) then
7:         clocks, touchedBlocksList = getClocksOfAllOpt3Paths(bb)
8:         if isClockable(avg, clocks) then
9:             setClock(bb, avg)
10:            for all tb in touchedBlocksList do
11:                removeClock(tb)
12:            end for
13:            tbSuccList = getAllSuccessorsOfTB(touchedBlocksList)
14:            for all succ in tbSuccList do
15:                updateOpt3Clocks(succ)
16:            end for
17:            return
18:        end if
19:    end if
20:    succList = getAllSucessors(bb)
21:    for all succ in succList do
22:        updateOpt3Clocks(succ)
23:    end for
24: end function

25: function APPLYOPT3
26:     for all f in Program do
27:         visitedList.clear()
28:         updateOpt3Clocks(f.entry())
29:     end for
30: end function
```

Figure 11: Pseudocode for Optimization 3

more than one successors. This is because shifting clock to the upper block in this case will cause a larger divergence in clock. In this example, since **if.end21** is at higher loop depth than **if.then28**, we remove the clock from **if.end21** and add it to **if.then28**.

C. Optimization 3 (Averaging of Clocks)

This optimization is based on the fact that paths emanating from a block in a function could be matching close together in total clock values. One can imagine it as a specialized case of the Optimization 1 (Function Clocking). For Function Clocking, we just considered the paths emanating from the entry block, but here we also check for paths besides the entry block. When forming paths for a block, we only consider blocks dominated by it (execution must pass through the dominating block to reach its dominated blocks).

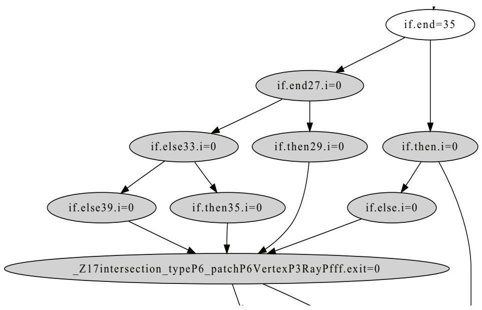

Figure 12: Part of example function after applying Optimization 3

The pseudocode for this optimization is shown in Figure 11. When finding paths for a block, we stop when we see backedges or when we see blocks with unclocked function calls. Moreover, we stop at a merge node if any of its successor is not dominated by the block in question. Like Optimizations 2a and 2b, we start to search for this optimization from the entry block of a function. If we find a block whose paths can be averaged, then after removing the clocks from the blocks in its path, we start to look for other blocks in the function. For this we consider the successors of the blocks in the path (from which the clocks were removed), given that those successors are not within that path. This is done by using the code from line 13 to 16 in Figure 11.

The example function after applying this optimization is shown in Figure 12. In this figure, accumulated clocks for all different four paths emanating from **if.end** were 37, 38, 38 and 29, with a mean value of 35.5 and standard deviation of 4.36. Since the range here is 8 (37-29) and is less than $mean/2.5$ as well as the standard deviation is 4.36, which is less than $mean/5$, we assign a clock of 35 to **if.end**, while removing clocks from all the blocks in the path. Note that we did not consider nodes below the merge node **_Z17intersection_typeP6_patchP6...** because it has **for.inc** as its successor, which is not dominated by **if.end**.

D. Optimization 4 (Loops)

This optimization considers the fact that loops are often executed multiple times. So for example, if you have a *for* loop, the increment operation will take place just before the next iteration. Therefore we check for back edges and if we see that the clock of the block from which the backedge is originating is less that a certain threshold value and is also less than the clock of the block it is jumping to, we merge its clock value to that block's clock and remove clock updating code from it. In this example, the clock of **for.inc** is merged with **for.cond**.

Figure 13 shows the example function after applying this final optimization.

Table I: Performance results of our scheme for the selected benchmarks

Benchmark	**Ocean**	**Raytrace**	**Water-nsq**	**Radiosity**	**Volrend**	**Average**
Original Exec Time	2903	670	1451	496	1340	-
Locks/sec	343	227835	126034	2211621	443070	-
Clockable Functions	1	33	1	39	35	-
After Inserting Clocks						
With No Optimization	2918 (1%)	718 (7%)	2082 (43%)	698 (41%)	1446 (8%)	20%
With Function Clocking Only (O1)	2901 (0%)	706 (5%)	2072 (43%)	644 (30%)	1445 (8%)	17%
With Conditional Blocks Optimization Only (O2)	2889 (0%)	715 (7%)	1779 (23%)	643 (30%)	1392 (4%)	13%
With Averaging of Clocks Only (O3)	2898 (0%)	702 (5%)	2072 (43%)	675 (36%)	1445 (8%)	18%
With Loops Optimization Only (O4)	2903 (0%)	707 (6%)	1752 (21%)	677 (36%)	1442 (8%)	14%
With All Optimizations	**2895 (0%)**	**695 (4%)**	**1748 (20%)**	**562 (13%)**	**1386 (3%)**	**8%**
After Inserting Clocks and Performing Deterministic Execution						
With No Optimization	2918 (1%)	768 (15%)	2096 (44%)	855 (72%)	1451 (8%)	28%
With Function Clocking Only (O1)	2924 (1%)	758 (13%)	2085 (44%)	711 (43%)	1448 (8%)	22%
With Conditional Blocks Optimization Only (O2)	2918 (1%)	766 (14%)	1785 (23%)	788 (57%)	1398 (4%)	20%
With Averaging of Clocks Only (O3)	2916 (0%)	742 (11%)	2090 (44%)	807 (63%)	1450 (8%)	25%
With Loops Optimization Only (O4)	2904 (0%)	760 (13%)	1761 (21%)	837 (69%)	1448 (8%)	22%
With All Optimizations	**2915 (0%)**	**742 (11%)**	**1758 (21%)**	**683 (38%)**	**1395 (4%)**	**15%**

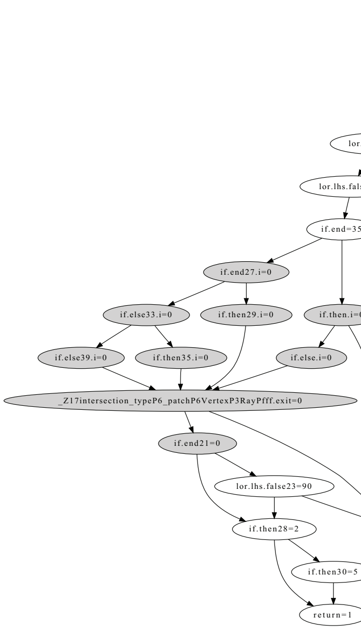

Figure 13: Example function after applying all optimizations

V. Performance evaluation

We selected only those benchmarks from SPLASH-2 [13] which only have locks and barriers as synchronization operations, as we have not yet implemented other synchronization operations, such as *condition variables* for example. All benchmarks were run on a 2.66 GHz quad core machine and compiled with maximum optimization enabled (level -O4 for clang/llvm). We first discuss the results. Afterwards, we show how clocking instructions ahead of time improves the deterministic execution. Lastly, we compare our results

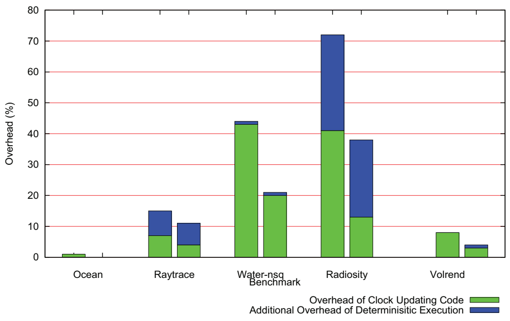

Figure 14: Overhead of inserting clocks and deterministic execution

with those from *Kendo*.

A. Results

Table I shows the performance overheads with different optimizations and Figure 14 gives a pictorial view of that overhead. The left bars in Figure 14 show the performance overhead without applying optimizations while the bars on the right show the overhead after applying all the optimizations. The lower portion of the bar is the overhead of the inserted clocks updating code only, while the upper portion shows the additional overhead for deterministic execution.

From Table I, we can see that different optimization affect different benchmarks differently. For example, Optimization 4 (Loops Optimization) has a significant impact on the performance of *Water-nsq* while not having that much effect on other benchmarks. This is because *Water-nsq* frequently executes a loop with a small body. The optimization that had the most impact on performance overall is Optimization 2 (Conditional Blocks Optimization). This is because conditional paths are frequently found in programs and this optimization efficiently reduces clock update for such paths. The Optimization 3 had the least impact. This is because it

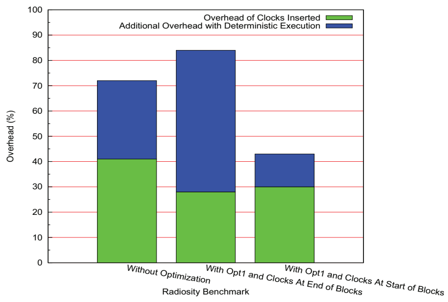

Figure 15: Improvement of the *Radiosity* benchmark from updating clocks ahead of time

is unlikely for a program to have all paths originating from a node to have similar clock values.

As far as Optimization 1 (Function Clocking) goes, *Radiosity* is one benchmark where this optimization significantly improved the performance. This is because this benchmark has such functions which are compute intensive and execute frequently. One interesting result of this optimization is that it significantly reduces the overhead of deterministic execution in addition to the reduction in clock updating overhead. This is discussed in more detail in the next section. Overall, we see that the average overhead of inserting clocks is at 8%, whereas the average overhead for deterministic execution is at 15%, with the overhead not exceeding 38% even for *Radiosity*, which has a very high lock frequency.

B. Effect of Updating Clocks Ahead of Time

There is a great benefit in updating the clock as soon as possible, so that threads waiting at a lock acquisition have to wait less. This effect is more evident for a benchmark like *Radiosity* which has high a lock frequency. From Table I, we can see that for *Radiosity*, although Optimization 2 (Conditional Blocks Optimization) reduces the clock overhead by the same amount as Optimization 1 (Function Clocking), Optimization 1 adds far less additional overhead for deterministic execution at 13% (43% - 30%) as compared to 27% (57% - 30%) in the case of Optimization 2. This is because Optimization 1 is able to increment the clock more aggressively ahead of time as it works for whole functions.

Figure 15 illustrates the effect of updating clocks ahead of time for *Radiosity*. Since *Function Clocking* optimization, where possible, increments the clock ahead of time the most (more than other optimizations), we only consider the result of Optimization 1 (Function Clocking) here. The left most bar is that without any optimization, the middle is with *Function Clocking* optimization, but clocks updated at the end of the basic blocks, whereas the right most bar is the same optimization but with clocks updated at the beginning of the basic blocks. From the figure, by looking at the upper portion of the bars, which represents the additional overhead of deterministic execution, we can see that updating clocks at the start of the block improves deterministic execution significantly as compared to updating them at the end.

C. Comparison with Kendo

In Table II, we compare our results with that of *Kendo*. Note that the purpose of our scheme is not to surpass *Kendo* in performance but to make it more portable while retaining sufficient efficiency. Since the data sets used by *Kendo* are not publicly available, neither its source code, we list the results directly from their paper. We tried to use the data sets which match the locks/sec frequency of those used by *Kendo*. For *Radiosity* and *Volrend*, we could not find matching data sets however and instead used data sets with higher lock frequencies than *Kendo*.

The only benchmark which performs worse than *Kendo* is *Water-nsq*. This is because *Water-nsq* executes a small *for* loop very frequently. The code inside that *for* loop contains an *if* statement. Although Optimization 2 (Conditional Blocks Optimization) and Optimization 4 (Loops Optimization) work to reduce overhead of clocks update in that loop, it still updates clocks frequently enough in that loop to have a relatively high overhead.

For *Radiosity*, which has a very high lock frequency, our scheme surpasses *Kendo* in performance. This is even when we used a data set which has a higher lock frequency than what *Kendo* used. This improvement in performance over *Kendo* can be explained by the fact that at compile time, we are able to update clocks before instructions are executed and thus reduce waiting time for a benchmark like *Radiosity* which has a high lock frequency. On the other hand, *Kendo* only updates the logical clocks when it receives overflow interrupts of the hardware performance counter that counts retired stores. Therefore, it cannot perform clock updates ahead of time. It also has to balance the chunk size of instructions executed between each interrupt, so as to reduce the impact of frequent interrupts while also maintaining frequent interrupts to keep the clocks incrementing. For *Radiosity*, the authors of *Kendo* had to manually adjust the chunk size to get the best performance, which is the one listed in Table II. Our scheme requires no such manual adjustments.

We even show slight improvement over *Kendo* for benchmarks which do not have very high lock frequencies, such as *Raytrace* and *Volrend*. This improvement can be explained from the fact that our scheme updates the clock more frequently than *Kendo*. Although, in case of *Kendo* there may be a less overhead of updating the clocks, threads who are in the process of acquiring a lock and thus waiting for other threads' clocks to go past them, have to wait longer due to the slow update of the clocks. Moreover, our optimizations prefer to update the clock even before instructions are executed. So even when we are updating the clocks less frequently, it is not because we are delaying their

Table II: Performance results of our scheme as compared to *Kendo*

Benchmark	**Ocean**	**Raytrace**	**Water-nsq**	**Radiosity**	**Volrend**
Results for Kendo					
Locks/sec	279	216979	143202	939771	79612
Overhead	1%	18%	7%	53%	7%
Results for our scheme					
Locks/sec	343	227835	126034	2211621	443070
Overhead	0%	11%	21%	38%	4%

update, but because we (most of the time) already updated them ahead of time.

VI. Conclusion

In this paper, we described our tool *DetLock*, which consists of an LLVM compiler pass to insert code for updating logical clocks for *Weak Deterministic* execution. Since our scheme does not depend on any hardware or modification of the kernel, it is very portable. Moreover, we apply several optimizations to reduce the amount of code inserted for clock updating. Furthermore, since the algorithm for *Weak Determinism* that we use gives lock to the thread with minimum logical clock, we try to increment the clocks of threads as soon as possible so that threads waiting for locks have to wait less. We increment the clocks even before instructions are executed if possible. On average, the overhead of inserting clock updating code is only 8%, whereas the overall overhead including deterministic execution is 15% for selected benchmarks. This performance is comparable to *Kendo*, while providing more portability. In fact for some applications, *DetLock* can even surpass *Kendo* in performance.

Acknowledgment

This research has been funded by the projects Smecy 100230, iFEST 100203 and REFLECT 248976.

References

[1] http://dpj.cs.uiuc.edu.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[3] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 264–272, New York, NY, USA, 2005. ACM.

[4] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333 –334, feb. 2011.

[5] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, 2010.

[7] H. Mushtaq, Z. Al-Ars, and K. Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 12 –17, dec. 2011.

[8] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *In Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007.

[9] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44:97–108, March 2009.

[10] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.

[11] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*, oct 2011.

[12] V. Weaver and J. Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? In *The 3rd Workshop on Functionality of Hardware Performance Monitoring*, FHPM '10, Atlanta, Georgia, 2010.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[15] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.

[16] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.

Computing (2014) 96:1131–1147
DOI 10.1007/s00607-013-0370-9

Efficient and highly portable deterministic multithreading (DetLock)

Hamid Mushtaq · Zaid Al-Ars · Koen Bertels

Received: 22 March 2013 / Accepted: 5 November 2013 / Published online: 19 November 2013

Abstract In this paper, we present *DetLock*, a runtime system to ensure deterministic execution of multithreaded programs running on multicore systems. *DetLock* does not rely on any hardware support or kernel modification to ensure determinism. For tracking the progress of the threads, logical clocks are used. Unlike previous approaches, which rely on non-portable hardware to update the logical clocks, *DetLock* employs a compiler pass to insert code for updating these clocks, thus increasing portability. For 4 cores, the average overhead of these clocks on tested benchmarks is brought down from 16 to 2 % by applying several optimizations. Moreover, the average overall overhead, including deterministic execution, is 14 %.

Keywords Multicore · Multithreading · Determinism

Mathematics Subject Classification 68N01 · 68W10 · 68N20

1 Introduction

Single threaded programs are much easier to test, debug and maintain than their multithreaded counterparts. This is because the only source of non-determinism in them is

In this paper, we extended DetLock, whose paper was published in MuCoCos 2012 affiliated with SC12. In this journal paper, we further improve the performance of DetLock by applying several more optimizations. Furthermore, we evaluated the performance with several more benchmarks.

H. Mushtaq (✉) · Z. Al-Ars · K. Bertels
TU Delft, Delft, Netherlands
e-mail: h.mushtaq@tudelft.nl

Z. Al-Ars
e-mail: z.al-ars@tudelft.nl

K. Bertels
e-mail: k.l.m.bertels@tudelft.nl

interrupts or signals, which are rare. On the other hand, multithreaded programs have a more frequent source of non-determinism in the form of shared memory accesses. Due to this, multithreaded programs suffer from repeatability problems, which means that running the same program with the same input can result different outputs. This repeatability problem makes multithreaded programs hard to test and debug. Furthermore, it is also difficult to build fault tolerant versions of these programs. This is because fault tolerant systems usually depend upon replicas (identical copies of redundant processes) to detect errors.

If access to shared data is not protected by synchronization objects in a multithreaded program, we can have race conditions, which may produce unexpected results. Running a program with race conditions deterministically does not avoid the problem of having unexpected results, but just makes sure that the same results can be replicated.

The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [2], which has a maximum overhead of 11x for 8 cores. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [15]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. Two such approaches are Calvin [7] and DMP [4]. They use the same concept as *CoreDet* for deterministic execution but make use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, we can relax the requirements to improve efficiency. For example, Kendo [13] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without race conditions. The authors of *Kendo* call it *Weak Determinism*. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [12], *Weak Determinism* is sufficient for most well written multithreaded programs. Therefore, *DetLock* also only supports *Weak Determinism*.

The basic idea of *Kendo* is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, *Kendo* still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counter that is deterministic [16]. Secondly, *Kendo* needs modification of the kernel to allow reading from the hardware performance counters for deterministic execution.

To overcome portability issues faced by *Kendo*, our tool *DetLock* has a completely software-based approach of updating the logical clocks. The code for updating the clocks is inserted through an LLVM [8] compiler pass. Since, LLVM is a popular open source compiler framework available on many platforms, our approach is portable across a wide range of platforms. Moreover, it requires no modification of the kernel. We can sum up the contribution of this paper as follows.

– A portable mechanism to update logical clocks for *Weak Deterministic* execution that depends upon the compiler rather than using hardware performance counters, since many platforms have no such deterministic counters available.
– A user-space approach to update the logical clocks that does not require modifying the kernel.
– A number of optimization techniques to reduce the overhead of the code used to update the logical clock and improve the performance of deterministic execution.

This paper is an extension on our previous work on this topic [11]. In this paper, we apply several more optimizations to improve the performance. This paper is organized as follows. In Sect. 2, we discuss the background and related work, while in Sect. 3, we give an overview of DetLock's architecture. This is followed by Sect. 4 where we present the optimization methods used to improve the performance of *DetLock*. In Sect. 5, we evaluate the performance of our scheme, and we finally conclude the paper with Sect. 6.

2 Background and related work

In this section, first we will discuss the state of the art for deterministic execution and then discuss our contribution.

2.1 State of the art

Single threaded programs are mostly deterministic in behavior. We say mostly because interrupts and signals can introduce non-determinism even in single threaded programs. However, these non-deterministic events are rare. On the other hand, in multithreaded programs running on multicore processors, shared memory accesses are a frequent source of non-determinism.

One way to ensure determinism of multithreaded programs is to write code for them in a deterministic parallel language. Examples of such languages are StreamIt [14] and SHIM [5]. The disadvantage of this approach is that porting programs written in traditional languages to deterministic languages is difficult as the learning curve is high for programmers used to programming in traditional languages. Moreover, in languages which are based on the Kahn Process Network Model, such as SHIM, it is difficult to write programs without introducing deadlocks [10].

Deterministic execution at runtime can be done either through hardware or software. Calvin [7] is a hardware approach that executes instructions in the form of chunks and later commits them at barrier points. It uses a relaxed memory model, where instructions are committed in such a way that only the total store order (TSO) of the program has to be maintained. DMP [4] uses a similar relaxed memory approach. The disadvantage of hardware approaches is that they are restricted to the platforms they were developed for.

Besides hardware methods, software only methods for deterministic execution also exist. One such method is CoreDet [2] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Therefore, it is similar to Calvin, but implemented in software. Logical clocks are used for deterministic execution. Since CoreDet is implemented in software, it has a very high overhead, possibly upto 11x for 8 cores, as compared to the maximum 2x for Calvin. Another similar approach is DTHREADS [15]. It runs threads as separate processes, so that memories which are modified can be tracked down through the memory management unit. Only at synchronization points such as locks, barriers and thread creation for example, it updates the shared memory from the local memories of the threads. Therefore, it avoids the overhead of using bulk synchronous quantas like CoreDet and also does not have the need to maintain logical clocks like CoreDet. However, the overhead for programs with high lock frequency or large memory usage is still very high.

Since performing deterministic execution in software alone is inefficient, Kendo [13] relaxes the requirements by only working for programs without race conditions (*Weak Determinism*). It does not use any hardware besides deterministic hardware performance counters found in some processors. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its clock becomes less than those of the other threads, with ties broken with thread IDs. Clock is calculated from retired stores, is paused when waiting for a lock and resumed after the lock is acquired. Kendo still suffers from portability problems as it requires hardware performance counters which are deterministic. Many platforms, including many x86 platforms, do not have any deterministic hardware performance counter [16]. Moreover, *Kendo* requires modification of the kernel to read from such hardware performance counters. A technique related to deterministic multithreading is record/replay. Examples of systems using this technique are Rerun [6], Karma [1] and Respec [9].

2.2 Our contribution

As discussed in the previous section, we already have tools such as *Kendo* to execute multithreaded programs deterministically on multicore platforms. However, one main bottleneck of using *Kendo* is that it requires deterministic hardware performance counters, which are not available on many platforms. For evaluation of their tool, the authors of *Kendo* had to specifically use the Core 2 processor, which had deterministic retired stores counters available on it. As we can see from Fig. 1, which shows the retired stores difference compared to the expected value, none of the listed processor besides Core 2, has a deterministic retired stores counter. Moreover, *Kendo*, also requires modification of the kernel to access these performance counters.

Now imagine a scenario, where a company wants to reduce the cost of testing for its software as well as ease maintainability of it by making it deterministic. If they go for the *Kendo* technique, it would make their software non-portable as it would be unable to run on processors which do not have any deterministic hardware performance

Fig. 1 Determinism of retired stores performance counter of various processors [16]

Machine	Before Adjustment	Adjusted
Core2	0±0	0±0
Atom	—	—
Nehalem	411,408±4	9±1
Nehalem-EX	411,914±6	9±1
Pentium D	163,402,604±185	11,776±175
Phenom	—	—
Istanbul	—	—

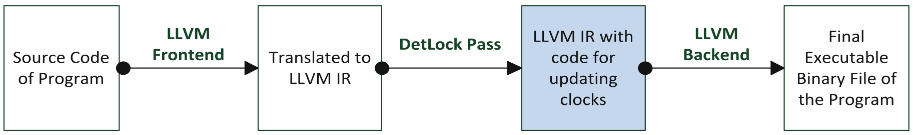
Source Code of Program
LLVM Frontend
Translated to LLVM IR
DetLock Pass
LLVM IR with code for updating clocks
LLVM Backend
Final Executable Binary File of the Program

Fig. 2 *DetLock* modifies the LLVM IR code by inserting code for updating logical clocks

counter. Moreover, you can expect users not wanting to modify the kernels of their operating systems. This is where our technique is useful, as it would allow a program to run on every machine, without requiring to modify the kernel.

This work is an extension of our previous work [11] on this topic. By applying new optimizations, we were able to further reduce the overhead of clock updating code inserted by our compiler pass, and improve performance of deterministic execution.

3 Overview of the architecture

In this section, we discuss the architecture of *DetLock* and the application programming interface (API) that it provides to the programmer.

3.1 Architecture

We use Kendo's algorithm to perform deterministic execution. However, unlike *Kendo* which requires deterministic hardware performance counters, which are not available on many platforms, we insert code to update logical clocks at compile time. This also means that we do not need to modify the kernel which is required by *Kendo* to read from performance counters. Figure 2 shows the point of compilation where the *DetLock* pass executes, which is between the point where the LLVM IR (intermediate representation) code is translated to the final binary code by the LLVM backend.

The unit of our logical clock is one instruction. For instructions which take more than one clock cycles, the logical clock is updated according to the approximate number of clock cycles they take. However, to keep our discussion simple, in this paper, for *DetLock* one instruction equals one logical clock count.

The Kendo's method of acquiring locks deterministically is illustrated in Fig. 3. In this figure, an example is given for a process with two threads. If Thread 1 is trying to acquire a lock when its logical clock is 1,029, it will not be able to do so if Thread

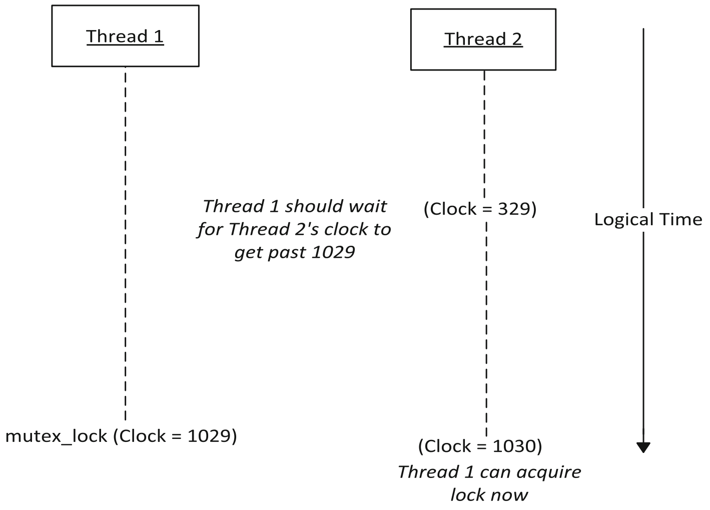
Thread 1
Thread 2
Thread 1 should wait for Thread 2's clock to get past 1029
(Clock = 329)
Logical Time
mutex_lock (Clock = 1029)
(Clock = 1030)
Thread 1 can acquire lock now

Fig. 3 Kendo's method of acquiring locks for deterministic execution

2's clock is at 329, because of being less than 1,029. But, as soon as Thread 2's clock get past 1,029, Thread 1 will acquire the lock.

So basically our purpose is not only to reduce the code that updates the clocks but also to update the clocks as soon as possible. In fact, at compile time it is possible to increment the clock even before instructions are executed. For example, if we know that a leaf function (a function with no function calls) executes fixed amount of instructions, we can increment the logical clock before executing any instruction of that function.

Therefore in all optimizations we apply, besides trying to reduce the clock update overhead, we also try to increment the clock as soon as possible. Without any optimization, we update the clock at start of each of the basic block of LLVM IR. If there is a function call inside that block, we split that block, such that each block either contains no function call or starts and ends with a function call. Then we update the clock at the top of each block if that block contains no function calls, otherwise we update the clocks in between the function calls. By splitting blocks in such a way, we can more easily apply optimizations.

3.2 Application programming interface

We provide our own functions for locks, barriers and thread creation for deterministic execution. They internally use the *pthread* library. However, it is not necessary for the programmer to modify the code to use them. A header file is provided by us that replaces the definition of these functions with ours. The header file can be specified in the makefile, thus making it unnecessary to modify source code files. Moreover, the code to initialize the clock for the main thread is inserted by the compiler.

It has to be noted that since our method depends upon the compiler to insert clocks for deterministic execution, it is not possible to increment the clocks in functions which are implemented in a library (since they have not been compiled with our pass). This problem also exists for functions which are built in the compiler, as LLVM generates no code for them at IR level. For many built-in functions such as *memset* and math functions, we just keep an estimate of the instructions they take and increment the clock accordingly. For *memset* and other functions which depend upon the size parameter, we increment the clock considering the size parameter. Since most built-in functions are simple, we can use an estimate for them. We provide a text file (instructions estimate file) for such purpose, where these functions can be defined with the approximate number of instructions they take along with their dependency on input parameters. However, this is not always possible for functions in shared libraries. One way is to ignore them and the other way is to add them in the *instructions estimate file* if possible (if the instructions count for them can be approximated satisfactorily).

Another concern are functions which internally use locks, such as *malloc*. For such functions, we provide our own implementation which replaces the locks with our own deterministic locks.

4 Performance optimizations

We apply several optimizations to reduce the clock updating overhead. Moreover, we try to increment clocks as soon as possible so that waiting time for threads who are waiting for other threads' clocks to go past them is reduced. Clock updating code is removed from the blocks whose clocks are made zero by our optimizations. In this paper, we highlight such blocks with gray color. To illustrate the effect of our optimizations, we are going to show how the optimizations change example functions. The clocks associated with each block are shown at the right of the assignment operators. Moreover, a block in parallelogram shape implies that it contains one or more function calls. The optimizations are discussed below.

4.1 Optimization 1 (function clocking)

As discussed in Sect. 3.1, the sooner the clocks are updated, the better, and leaf functions with only one basic block are perfect candidates for such an optimization. Clocks can be removed from such functions and instead be added to the basic blocks calling such functions. Besides functions with only one blocks, our method also considers leaf functions with multiple blocks, given that there are no loops in such functions. If our pass sees that all possible paths taken by such a function do not differ by much, we calculate the mean value for all possible paths and use that mean value to update the clock. The criteria we have set is that the minimum and maximum clock difference of all possible paths should not be more than the mean value divided by 2.5. Moreover the standard deviation between all the different paths should not be greater than one fifth of the mean value.

We call such leaf functions as clocked functions. By intuition, we can judge that it is also possible to clock functions which call only clocked functions. In this way, we can even clock functions which are not necessarily leaf functions. More detail on this optimization can be found in [11].

Previously, in [11], we did not consider the possibility of clocked functions being called indirectly through functions pointers. In that case, since we remove the clocks

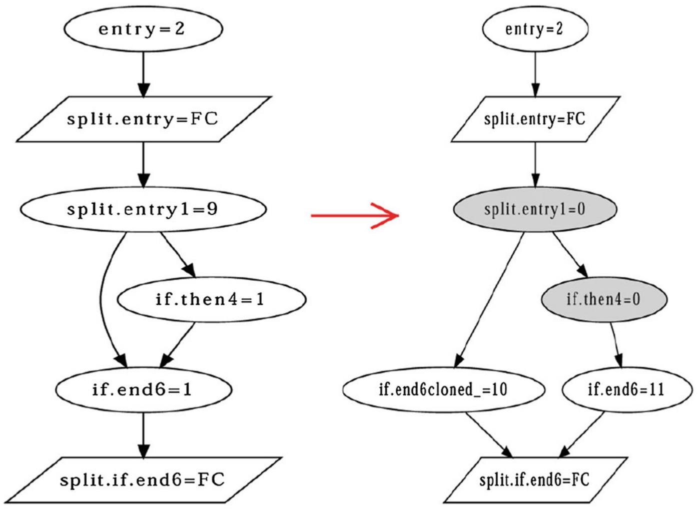

Fig. 4 Example function before and after applying optimization 2b

of clocked functions, the clock does not get updated. To correct this problem, we create clones of clocked functions. The clock from cloned clock functions is removed but clock updating code is inserted in the original clocked functions. Wherever our pass finds a direct call of a clocked function, it replaces it with call to the cloned version of that clocked function. However, since indirect calls still call the original clocked function, the clock does get updated properly even with indirect calls.

4.2 Optimization 2 (conditional blocks)

This optimization deals with if-else and switch statements and consists of four parts, which are described below.

4.2.1 Opt 2a: pushing clocks upwards

This optimization is based on the principle that if a block has two or more successors, we can make the successor with the least clock zero and subtract its original value from all its siblings, while also adding its original clock to the parent block. Another principle of this optimization is that if all predecessors of a merge block have that merge block as their only successor, the clocks could be shifted from the merge node to them. More details about this optimization can be found in [11].

4.2.2 Opt 2b: cloning blocks

In this optimization, we clone blocks where possible to reduce the number of clock updates. For example, for the example function shown in Fig. 4, block *if.end6* is cloned, so that for the paths formed by blocks *split.entry1*, *if.then4* and *if.end6*, clock needs to be updated only once, rather than twice or thrice.

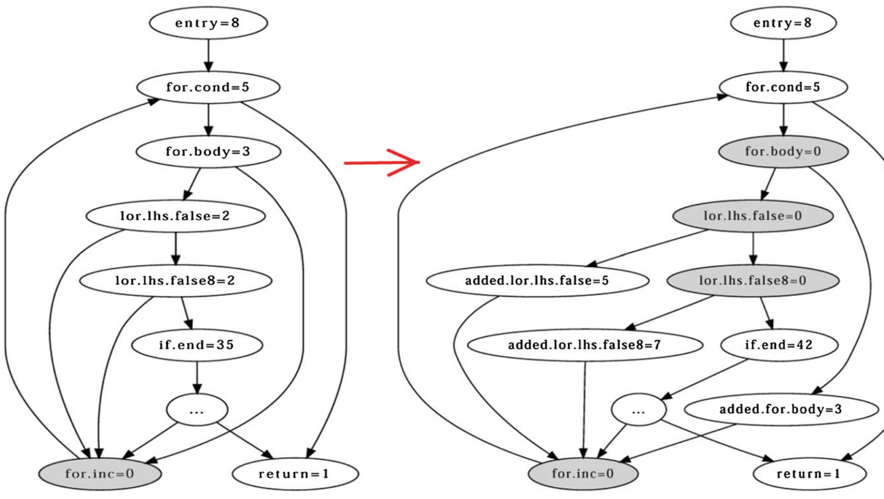

Fig. 5 Example function before and after applying optimization 2c

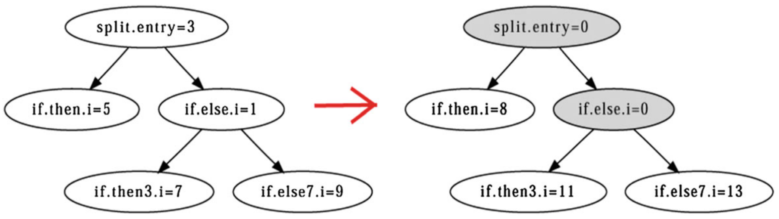

Fig. 6 Part of an example function before and after applying optimization 2d

4.2.3 Opt 2c: adding additional blocks

In this optimization, we add new blocks where necessary to reduce the clock updating code. To illustrate this, an example function before and after applying this optimization is shown in Fig. 5. Three new blocks are added to update the clocks removed from the three blocks shown in gray in the optimized version. The accumulated clock of those three blocks is also added to the clock of block *if.end*. With this optimization, for the path from *for.cond5* to *if.end*, clock is updated only twice, while in the unoptimized version it is updated 5 times. Note that the block (. . .) here represents a bundle of basic blocks, which we do not show due to space limitations.

4.2.4 Opt 2d: pushing clocks downwards

In this optimization, we update the clock from top to bottom. This optimization can remove clocks more efficiently in some cases as compared to *Opt 2a*. However, Since we try to update clocks as soon as possible, we apply this optimization only for paths where the accumulated clock is less than a certain value. Part of an example function, before and after applying this optimization is shown in Fig. 6. We can see that with this optimization, for the part of the example function, clock is only updated once, rather than twice or thrice in the original version of that function.

4.3 Optimization 3 (averaging of clocks)

This optimization is based on the fact that paths emanating from a block in a function could be matching close together in total clock values. One can imagine it as a specialized case of the optimization 1 (function clocking). For function clocking, we just considered the paths emanating from the entry block, but here we also check for paths besides the entry block. When forming paths for a block, we only consider blocks dominated by it (execution must pass through the dominating block to reach its dominated blocks). More details about this optimization can be found in [11].

4.4 Optimization 4 (loops)

This optimization deals with loops. The different types of optimization we applied on loops are discussed next.

4.4.1 Opt 4a: forwarding clocks from blocks with backedges

This optimization considers the fact that loops are often executed multiple times. So for example, if you have a *for* loop, the increment operation will take place just before the next iteration. Therefore we check for back edges and if we see that the clock of the block from which the backedge is originating is less than a certain threshold value and is also less than the clock of the block it is jumping to, we merge its clock value to that block's clock and remove clock updating code from it.

4.4.2 Opt 4b: incrementing clocks before for loops

This optimization is based on the fact that for many *for* loops, the number of iterations can be checked at compile time. So for example, if at compile time, we can see that a *for* loop is executed n number of times, we can update the clock ahead of time. First our pass figures out the least number of instructions an iteration in a loop will execute. This number multiplied with n is incremented before execution of the *for* loop. Inside the *for* loop, we update the clock only where it is necessary.

An example function before and after applying this optimization is shown in Fig. 7. Here, the minimum number of instructions executed by the *for* loop for an iteration is 21. Therefore it is multiplied by the number of iterations N of the *for* loop.

The pseudocode for optimization 4b is shown in Fig. 8. For each block in a function, it checks if its a loop header. This optimization is only applied for inner most loops, as they are usually the most compute intensive types of loops. The *meetsOpt4bRequirements* checks if all blocks inside the loop are at the same level, as well as checks other things, such as, no block has unclocked functions. If all the requirements are met, the optimization is applied. *preds* here are the predecessor blocks of the loop header. For example, in the example function shown in Fig. 7, block

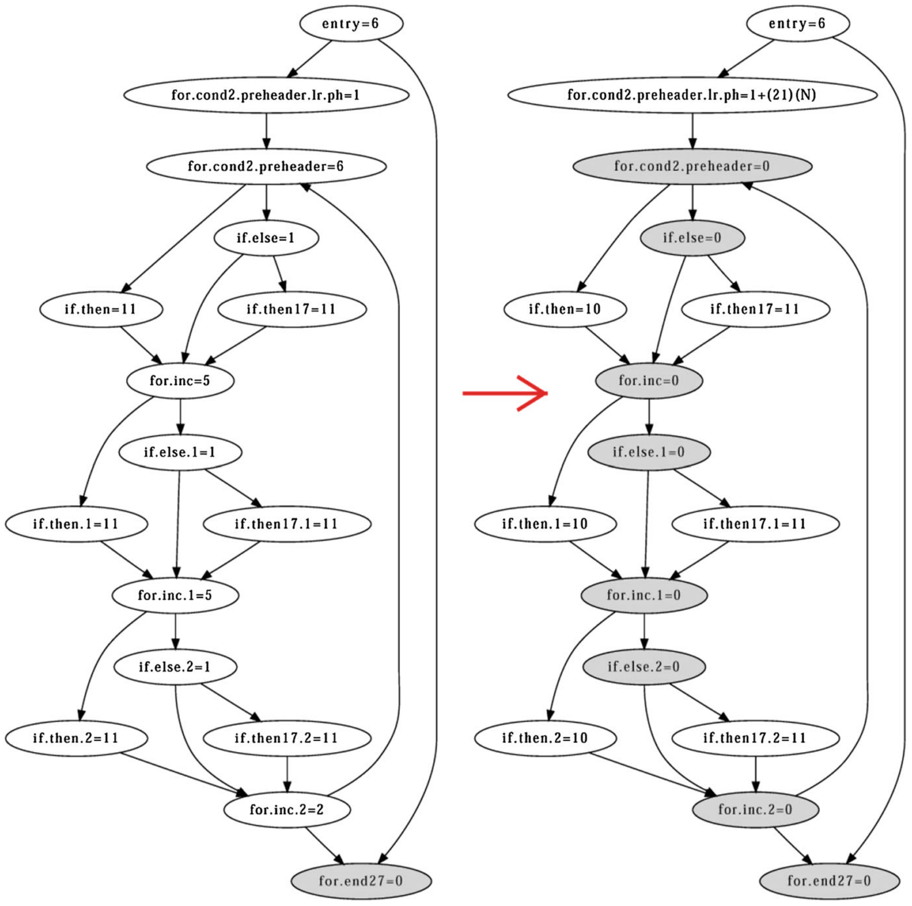

Fig. 7 Example function before and after applying optimization 4b

for.cond2.preheader.lr.ph is the predecessor. If there is only one predecessor, which have no other successor than the loop header, the clock is added to that predecessor block, as shown by the code on line 14 to 19. Otherwise, a block is added in between the loop header and its predecessors to update the clock. The *be* block returned by *meetsOpt4bRequirements* is the block that contains the backedge, which is *for.inc.2* in this case. This is passed as a parameter to the *updateClocksInLoop* function, which shifts the constant clock value of the loop, that is, the least number of instructions the loop will always execute in an iteration, to the header block. The value in the header block is then shifted to the predecessor block. In this case, the *updateClocksInLoop* function added 15 to the original clock of *for.cond2.preheader*, which is later shifted to the predecessor block *for.cond2.preheader.lr.ph*.

UpdateClocksInLoop works by first concentrating clock to all merge nodes which are guaranteed to be executed in an iteration. Such blocks are *for.inc*, *for.inc1* and *for.inc2* in the example function. The list of all blocks preceding such a merge node is checked to see which one has minimum value, and that minimum value is added

```
function updateOpt4bClocks(ref bool modified, ref BasicBlock bb)    ▷ When this
function is called Entry block of a function is passed as bb
    if visitedList.find(bb) then
        return
    end if
    visited.insert(bb)
    modified = false
    preds, be, loopIterInfo, meetsReq = meetsOpt4bRequirements(bb)
    if meetsReq then
        pathsList, blocksInLoop = getLoopPaths(bb, be)
        blocksPresentInAllPaths = getBlocksPresentInAllPaths(pathsLists, bb)
        if blocksInLoop.size() > 1 then
            updateClocksInLoop(blocksPresentInAllPaths, bb, be, blocksInLoop)
        end if
        loopIterInfo.constLoopClock = getClock(bb)
        setClock( bb, 0 )
        if preds.size() > 1 or numSuccessors(preds[0]) > 1 then
            addedBlocks = addBlockInBetween(preds, bb)
            setloopIterInfo(addedBlocks, loopIterInfo)
        else
            setloopIterInfo(preds[0], loopIterInfo)
        end if
    end if
    succList = getSuccList(bb)
    for all succ in succList do
        updateOpt4bClocks(modified, succ)
    end for
end function

function ApplyOpt4b
    for all f in Program do
        modified = true
        while modfied do
            visitedList.clear()
            updateOpt4bClocks(modified, f.entry())
        end while
    end for
end function
```

Fig. 8 Pseudocode for optimization 4b

to such merge node. For example, *if.then* and *if.else* are blocks preceding the merge node *for.inc*. *if.then17* is not included in the list since its being dominated by *if.else*, which is already preceding the merge node in question. Since *if.else* has the minimum clock of the two, its clock (1) is added to *for.inc*, making clock of *for.inc* 6, while making clock of *if.else* 0, and also subtracting 1 from *if.then*. The same is done with the other two merge nodes. After doing this step, *for.inc* and *for.inc1* contain clock values of 6 each while *for.inc2* contain clock value of 3. So, overall these three merge nodes contain clock value of 15. This clock value is removed from all these nodes and shifted to *for.cond2.preheader* to make its clock 21 from 6, while making the clocks of these merge nodes 0. Later on, clock is removed from *for.cond2.preheader* and updated before executing the loop in *for.cond2.preheader.lr.ph* by multiplying it with the number of times the loop will execute.

4.4.3 Opt 4c: cloning while loops

It is not possible to apply optimization 4b, where the number of iterations of a loop could not be determined at compile time, which is usually the case with *while* loops. If

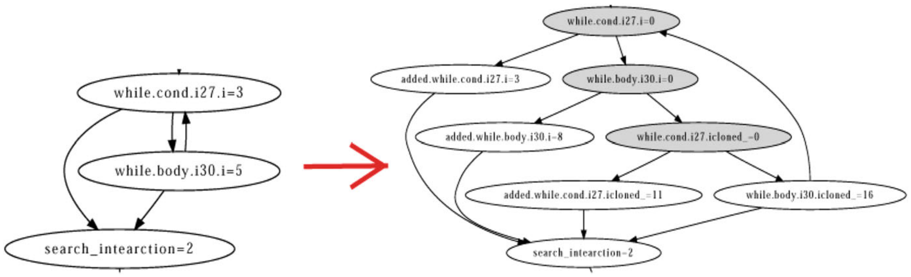

Fig. 9 Part of an example function before and after applying optimization 4c

a *while* loop executes constant number of instructions in each iteration, we clone that while loop, so that clock is updated at every other iteration rather than each iteration. A part of an example function before and after applying this optimization is shown in Fig. 9. Note that even if we applied optimization *4a* here, the clock would have been updated after 8 instructions. But by cloning these loops, it is executed after 16 instructions. Note that we also add blocks to update the clock if the *while* loop exits on an odd iteration.

5 Performance evaluation

We tested performance of *DetLock* with 8 benchmarks, 6 from SPLASH-2 [17] and 2 from PARSEC [3]. All benchmarks were run on a 2.66 GHz quad core machine and compiled with maximum optimization enabled (level -O4 for clang/llvm). We first discuss the results. Afterwards, we show how clocking instructions ahead of time improves the deterministic execution.

5.1 Results

Table 1 shows the performance overheads with different optimizations and Fig. 10 gives a pictorial view of that overhead. In Table 1, along with the results with different optimizations, we also show the original execution times, locks per second and number of clocked functions for each benchmark. Note that all the times are in milliseconds. The left bars in Fig. 10 show the performance overhead without applying optimizations, while the bars in the middle show performance after applying optimizations of [11] only. The bars on the right show the overhead after applying all the optimizations, including those mentioned in this paper. The lower portion of the bar is the overhead of the inserted clocks updating code only, while the upper portion shows the additional overhead for deterministic execution.

From Fig. 10, we can see that new optimizations introduced in this paper improved performance for several benchmarks as shown by the decrease in size of the right bars. The improvement relative to [11] is most significant for *barnes*, *water* and *swaptions*. For *water* for example, the overhead of deterministic execution is brought down from 20 to 0 %. Table 1 show performance with different optimizations. The optimization 4

Table 1 Performance results of our scheme for the selected benchmarks

BM	Barnes	Ocean	Volrend	Water	Swaption	Fluidanim	Raytrace	Radiosity	Avg
Orig	1,191	2,656	949	1,333	892	1,447	2,165	928	–
Locks/sec	462,101	375	125,293	141,067	689,283	2,472,882	241,757	3,170,433	–
Clk Funcs	9	0	34	0	2	0	32	39	–
After inserting clocks									
No Opt	1,436 (21 %)	2,681 (1 %)	975 (3 %)	1,890 (42 %)	959 (8 %)	1,701 (18 %)	2,307 (7 %)	1,225 (32 %)	1,647 (16 %)
Only [11]	1261 (6 %)	2,669 (0 %)	950 (0 %)	1,603 (20 %)	952 (7 %)	1,530 (6 %)	2,305 (6 %)	969 (4 %)	1,530 (6 %)
Opt 1	1,361 (14 %)	2,681 (1 %)	973 (3 %)	1,890 (42 %)	949 (6 %)	1,701 (18 %)	2,306 (7 %)	1,148 (24 %)	1,626 (14 %)
Opt 2	1,242 (4 %)	2,667 (0 %)	950 (0 %)	1,387 (4 %)	954 (7 %)	1,638 (13 %)	2,306 (7 %)	1,013 (9 %)	1,520 (6 %)
Opt 3	1,257 (6 %)	2,672 (1 %)	950 (0 %)	1,880 (41 %)	953 (7 %)	1,701 (18 %)	2,307 (7 %)	1,089 (17 %)	1,601 (12 %)
Opt 4+1	1,271 (7 %)	2,657 (0 %)	951 (0 %)	1,333 (0 %)	917 (3 %)	1,688 (17 %)	2,296 (6 %)	1,109 (20 %)	1,528 (6 %)
All Opts	1,199 (1 %)	2,656 (0 %)	949 (0 %)	1,333 (0 %)	913 (2 %)	1,527 (6 %)	2,247 (4 %)	968 (4 %)	1,474 (2 %)
After inserting clocks and performing deterministic execution									
No Opt	1,547 (30 %)	2,698 (2 %)	1,036 (9 %)	1,890 (42 %)	1,154 (29 %)	2,411 (67 %)	2,412 (11 %)	1,557 (68 %)	1,838 (32 %)
Only [11]	1,403 (18 %)	2,675 (1 %)	951 (0 %)	1,604 (20 %)	1,076 (21 %)	2,220 (53 %)	2,403 (11 %)	1,249 (35 %)	1,698 (20 %)
Opt 1	1,403 (18 %)	2,698 (2 %)	1,034 (9 %)	1,890 (42 %)	1,151 (29 %)	2,411 (67 %)	2,411 (11 %)	1,354 (46 %)	1,794(28 %)
Opt 2	1,449 (22 %)	2,682 (1 %)	951 (0 %)	1,388 (4 %)	1,148 (29 %)	2,315 (60 %)	2,412 (11 %)	1,454 (57 %)	1,725 (23 %)
Opt 3	1,436 (21 %)	2,676 (1 %)	972 (2 %)	1,880 (41 %)	1,112 (25 %)	2,364 (63 %)	2,411 (11 %)	1,437 (55 %)	1,786 (27 %)
Opt 4+1	1,369 (15 %)	2,668 (0 %)	984 (4 %)	1,334 (0 %)	1,008 (13 %)	2,396 (66 %)	2,390 (10 %)	1,353 (46 %)	1,688(19 %)
All Opts	1,211 (2 %)	2,667 (0 %)	950 (0 %)	1,333 (0 %)	1,005 (13 %)	2,220 (53 %)	2,385 (10 %)	1,247 (34 %)	1,627 (14 %)

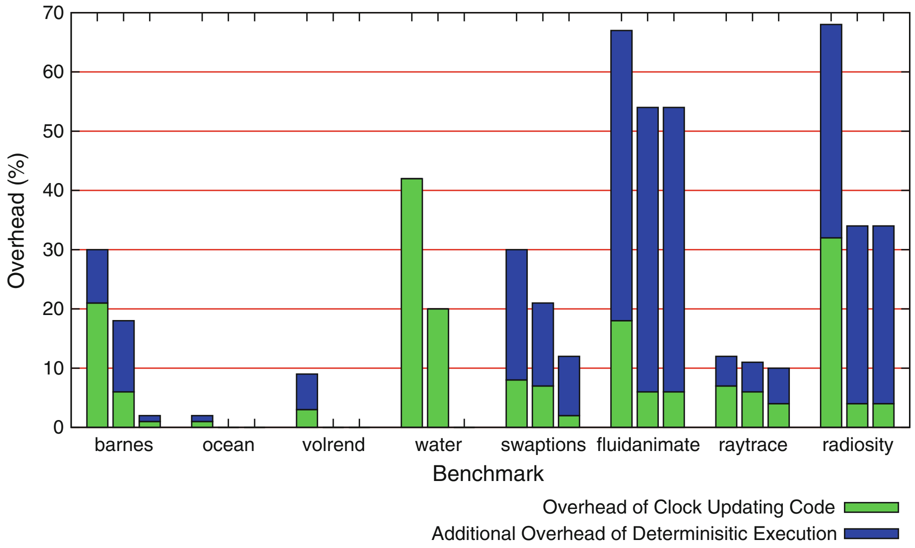

Fig. 10 Overheads

is shown combined with optimization 1 in the table, because some loops in the benchmarks consist of clocked functions, and without adding optimization 1, the impact of optimization 4 could not be seen in such cases. Overall, we see that all optimizations work to reduce the clock updating overhead as well as the deterministic execution overhead. However, from the *average* column, we can see that while optimization 2 reduced the clock update overhead more than optimization 4 and 1 combined, the later reduced the overall time, including deterministic execution more than optimization 2. The reason for this is discussed next.

5.2 Effect of updating clocks ahead of time

From Table 1, we can see that optimization 4 and 1 combined reduce the overall deterministic execution time more than optimization 2. This is even when optimization 2 reduced the clock updating overhead more than optimization 4 and 1 combined. The reason for this is because updating clocks ahead of time reduces waiting time of a thread which is in the process of acquiring a lock, as the clocks of other threads progresses more quickly in this way (even before execution of some instructions), thus allowing a waiting thread's clock to reach the minimum global more quickly. This effect is most pronounced for *swaptions* and *radiosity*. For *swaptions* for example, clock updating overhead with optimization 2 is 7 % while overall deterministic execution overhead with Optimization 2 is 29 %. On the other hand, with optimization 4 and 1 combined, the clock updating overhead is 3 % while overall deterministic overhead is only 13 %, which means an increase of only 10 % overhead over clock updating overhead as compared to an increase of 22 % with optimization 2. Similarly, for *radiosity*, there is an increase of 26 % (46–20 %) overhead over clock updating overhead with optimization

4 and 1 combined as compared to an increase of 48 % (57–9 %) with optimization 2. This is because optimizations 4 and 1 can more aggressively increment clock ahead of time as compared to optimization 2, as optimization 4 and 1 work at function and loop levels, whereas optimization 2 works only at basic blocks level.

6 Conclusion

In this paper, we described our tool *DetLock*, which consists of an LLVM compiler pass to insert code for updating logical clocks for *Weak Deterministic* execution. Since our scheme does not depend on any hardware or modification of the kernel, it is very portable. Moreover, we apply several optimizations to reduce the amount of code inserted for clock updating. Furthermore, since the algorithm for *Weak Determinism* that we use gives lock to the thread with minimum logical clock, we try to increment the clocks of threads as soon as possible so that threads waiting for locks have to wait less. We increment the clocks even before instructions are executed if possible. On average, the overhead of inserting clock updating code is only 2 %, whereas the overall overhead including deterministic execution is 14 % for selected benchmarks. This is an improvement over our previous work [11], with which on average, the overhead of inserting clock updating code is 6 %, while overall overhead including deterministic execution is 20 %.

Acknowledgments This research has been funded by the projects Smecy 100230, iFEST 100203 and REFLECT 248976.

References

1. Basu A, Bobba J, Hill MD (2011) Karma: scalable deterministic record-replay. In: ICS ’11. ACM, New York, p 359–368
2. Bergan T, Anderson O, Devietti J, Ceze L, Grossman D (2010) Coredet: a compiler and runtime system for deterministic multithreaded execution. SIGARCH Comput Archit News 38:53–64
3. Bienia C, Kumar S, Singh JP, Li K (2008) The parsec benchmark suite: characterization and architectural implications. In: PACT ’08. ACM, New York, p 72–81
4. Devietti J, Lucia B, Ceze L, Oskin M (2009) Dmp: deterministic shared memory multiprocessing. In: ASPLOS ’09. ACM, New York, p 85–96
5. Edwards SA, Tardieu O (2005) Shim: a deterministic model for heterogeneous embedded systems. In: EMSOFT ’05. ACM, New York, p 264–272
6. Hower DR, Hill MD (2008) Rerun: exploiting episodes for lightweight memory race recording. In: ISCA ’08. IEEE Computer Society, Washington, DC, p 265–276
7. Hower D, Dudnik P, Hill M, Wood D (2011) Calvin: deterministic or not? free will to choose. In: HPCA ’11. p 333–334
8. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO’ 04. Palo Alto
9. Lee D, Wester B, Veeraraghavan K, Narayanasamy S, Chen PM, Flinn J (2010) Respec: Efficient online multiprocessor replay via speculation and external determinism. In: ASPLOS’10. p 77–89
10. Mushtaq H, Al-Ars Z, Bertels K (2011) Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In: IDT ’11. p 12–17
11. Mushtaq H, Al-Ars Z, Bertels K (2012) DetLock: portable and efficient deterministic execution for shared memory multicore systems. In: MuCoCoS ’12, Salt Lake City
12. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 programming language design and implementation conference

13. Olszewski M, Ansel J, Amarasinghe S (2009) Kendo: efficient deterministic multithreading in software. SIGPLAN Not 44:97–108
14. Thies W, Karczmarek M, Amarasinghe SP (2002) Streamit: a language for streaming applications. In: CC ’02. Springer-Verlag, London, p 179–196
15. Tongping Liu EDB, Curtsinger Charlie (2011) Dthreads: efficient deterministic multithreading. In: SOSP ’11
16. Weaver V, Dongarra J (2010) Can hardware performance counters produce expected, deterministic results? In: FHPM ’10. Atlanta
17. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The splash-2 programs: characterization and methodological considerations. SIGARCH Comput Archit News 23:24–36

Fault Tolerance on Multicore Processors using Deterministic Multithreading

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

***Abstract*—This paper describes a software based fault tolerance approach for multithreaded programs running on multicore processors. Redundant multithreaded processes are used to detect soft errors and recover from them. Our scheme makes sure that the execution of the redundant processes is identical even in the presence of non-determinism due to shared memory accesses. This is done by making sure that the redundant processes acquire the locks for accessing the shared memory in the same order. Instead of using record/replay technique to do that, our scheme is based on *deterministic multithreading*, meaning that for the same input, a multithreaded program always have the same lock interleaving. Unlike record/replay systems, this eliminates the requirement for communication between the redundant processes. Moreover, our scheme is implemented totally in software, requiring no special hardware, making it very portable. Furthermore, our scheme is totally implemented at user-level, requiring no modification of the kernel. For selected benchmarks, our scheme adds an average overhead of 49% for 4 threads.**

I. Introduction

The abundant computational resources available in multicore systems have made it feasible to implement otherwise prohibitively intensive tasks on consumer grade systems. However, these systems integrate billions of transistors to implement multiple cores on a single die, thus raising reliability concerns, as smaller transistors are more susceptible to both transient [10] as well as permanent [11] faults.

A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the state machine replication approach [12]. In this approach the replicated copies of a process (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions (such as *gettimeofday*) deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the replicas perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

One way of making sure that the redundant processes access the shared memory in the same order is to perform record/replay where the leader process records the order of locks (to access shared memory) in a queue which is shared between the leader and follower. The follower in turn reads from that queue to have the same lock acquisitions order. This approach is used by Respec [4] and our previous work [6]. However, this requires communication between the leader and follower process, which decreases reliability, as the memory used for communicating might itself become corrupted due a soft error. Moreover, it requires extra memory.

In this scheme, instead of depending on record/replay, we use *deterministic multithreading*, that is, given the same input, a multithreaded process always have the same lock interleaving. This makes sure that the redundant processes acquire the locks in the same order without communicating with each other. We adapt the method used by Kendo [5] to do this, but unlike *Kendo*, our scheme neither requires deterministic hardware performance counters, which are not available on many platforms [8] (including many x86 systems), nor kernel modification for deterministic execution. The logical clocks used for deterministic execution are inserted by the compiler instead.

We can sum up the contributions of this paper as follows.

1) The scheme is implemented using a user-level library and does not require a modified kernel.
2) The scheme uses *deterministic multithreading* instead of record/replay to ensure that the redundant processes acquire locks for shared memory access in the same order. This eliminates the requirement of communication between replicas for deterministic shared memory accesses, making the system more reliable (by increasing isolation) and consume less memory.
3) The scheme is very portable since it does not depend upon any special hardware for determinisitic execution.

In Section II we discuss the background and related work, while in Section III, we discuss our fault tolerance scheme. This is followed by Section IV, where we discuss the implementation. In Section V, we evaluate the performance of our scheme, and we finally conclude the paper with Section VI.

II. Background and related work

A fault tolerant system which uses redundant execution needs to make sure that the redundant processes do not diverge in the absence of faults. In a single threaded program, in the absence of any fault, the only possible causes of divergence among the replicas can be non-deterministic functions (such as *gettimeofday*) or asynchronous signals/interrupts.

However, in multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses. These accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve.

One method to ensure redundant processes access shared memory in the same order is record/replay. Both software and hardware methods exist for that purpose. An example of hardware approach is Karma [14] which intercepts the cache coherence protocols to record inter-processor data dependencies and later use these recorded data dependencies to replay. Respec [4] is a software-based method. It logs the ordering of acquisition and release of synchronization objects, such as mutexes, to make replicas acquire the synchronization objects in the same order. It also performs checkpoint/rollback to perform recovery.

The disadvantage of employing record/replay for deterministic shared memory accesses is that it requires communication between the replicas, making the fault tolerant scheme less reliable as the shared memory used for communication can itself become corrupted by one of the replicas. Moreover it requires extra memory.

To eliminate this communication and memory requirement, we can employ *deterministic multithreading*, where a multithreaded process have always the same memory interleaving for the same input. The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions, that is, provide *strong determinism*. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [1], which has a maximum overhead of 11x for 8 cores. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [7]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. An example of such approach is Calvin [3], which uses the same concept as *CoreDet* for deterministic execution but make use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, one can relax the requirements to improve efficiency. For example, Kendo [5] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without

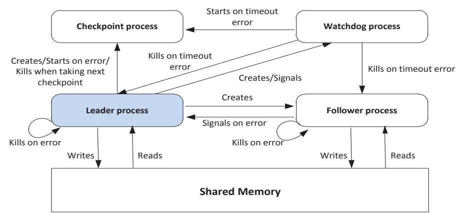

Fig. 1. Block diagram of our fault tolerance scheme

race conditions. The authors of *Kendo* call it *Weak Determinism*. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [15], *Weak Determinism* is sufficient for most well written multithreaded programs.

The basic idea of *Kendo* is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, *Kendo* still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counters that are deterministic [8]. Secondly, *Kendo* needs modification of the kernel to allow reading from the hardware performance counters.

III. FAULT TOLERANCE SCHEME

Our fault tolerant scheme is intended to reduce probability of failures in the presence of transient faults. The block diagram of our fault tolerance scheme is shown in Figure 1.

Initially, the leader process (which is the original process highlighted in the figure) creates the watchdog and follower processes. The follower process is identical to the leader process and follows the same execution path. The execution is divided into time slices known as epochs. An epoch starts and ends at a program barrier. At the end of each epoch, the memories of the leader and follower processes are compared. If no divergence is found, a checkpoint is taken and output to files or screen is committed. The previous checkpoint is also deleted. The checkpoint is basically a suspended process which is identical to the leader process at the time the checkpoint is taken. If a divergence is found at the end of an epoch, execution is restarted from the last checkpoint by resuming the checkpoint process and killing the leader and follower processes. This can also happen inside an epoch, if the follower sees that the parameters of system calls logged by the leader do not match those read by the follower, in which case it signals the leader process to restart execution from the last checkpoint. When the checkpoint process starts, it becomes the leader and creates its own follower. The watchdog process is used to detect timeout errors and recover from them. This is done by having the watchdog process signal the checkpoint process to start on a timeout error.

The approach used in this paper is different from *Respec* and our previous work [6] in the way it makes sure that leader

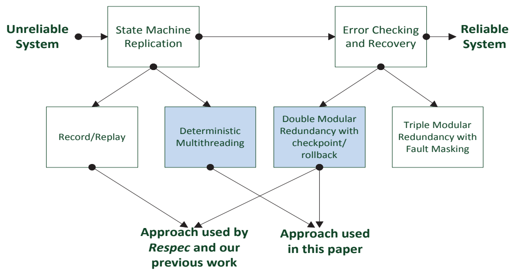
Unreliable System
State Machine Replication
Error Checking and Recovery
Reliable System
Record/Replay
Deterministic Multithreading
Double Modular Redundancy with checkpoint/ rollback
Triple Modular Redundancy with Fault Masking
Approach used by *Respec* and our previous work
Approach used in this paper

Fig. 2. Steps used by our fault tolerance scheme

and follower processes are identical in the presence of non-deterministic shared memory accesses. While *Respec* and our previous work used the record/replay technique, where the leader logs the synchronization operations (to access shared memory) in a queue which is then read by the follower to have the same order of synchronization operations, in this paper, we use *deterministic multithreading*, which does not require such type of communication, thus improving isolation and fault tolerance. The difference in these two approaches is illustrated in Figure 2. Note that in this approach, we still use shared memory between leader and follower, but only to log results from non-deterministic functions and type, input parameters and results of system calls, besides using it for memory comparison at the end of epochs for error checking.

At this moment, our fault tolerance scheme does not work with programs that use inter process communication (through pipes and shared memory for example). The only form of I/O allowed is disk I/O and screen output. Moreover, our scheme assumes that there are no data races in the program. Lastly, we have not added functionality to handle asynchronous signals. However, this functionality can be added for user space by handling asynchronous signals at synchronous points, such as system calls, as done by Scribe [13].

IV. Implementation

In this section, we discuss the implementation of our fault tolerance scheme. We start by Section IV-A, where we discuss how deterministic execution of the replicas is performed. This is followed by Section IV-B which discusses error detection. Finally in Section IV-C, we discuss our recovery mechanism.

A. Deterministic execution

For deterministic execution, we need to ensure that replicas use the same memory addresses. We also need to ensure determinism in the presence of non-deterministic functions and shared memory accesses. Moreover, we need to make sure that the leader and follower processes use the same memory addresses. For this we need to have a deterministic memory allocation scheme. Finally we also need to make sure that we have deterministic I/O. Below we discuss how we handle these issues.

1) Replica creation: Our library assumes that threads in the application are created once at the start of the application. Therefore, we create the follower process at point in the code where the threads are created. For this purpose, we replace the *pthread_create* function with our own to make sure the threads of the replicas use the same memory addresses. More detail on this can be found in [6].

2) Memory allocation: We implement our own memory allocation functions to allocate memory deterministically. Our implementation replaces the locks in the original memory allocation functions with our own deterministic locks. The variables used by our library (not related to original program execution) to perform deterministic execution, may have different values for the leader and follower processes, for example, the flag used to distinguish the leader process from the follower process. For these variables, we use a separate memory, which is allocated with the *mmap* system call. This memory is not compared for error detection.

3) Deterministic shared memory accesses: We use Kendo's algorithm to perform deterministic execution. However, unlike *Kendo* which requires deterministic hardware performance counters, which are not available on many platforms, we insert code to update logical clocks at compile time. This also means that we do not need to modify the kernel which is required by *Kendo* to read from performance counters. Figure 3 shows the point of compilation where our compiler pass executes, which is between the point where the LLVM IR (Intermediate Representation) code is translated to the final binary code by the LLVM backend.

The unit of our logical clock is one instruction. For instructions which take more than one clock cycles, the logical clock is updated according to the approximate number of clock cycles they take.

The Kendo's method of acquiring locks deterministically works by giving lock to the thread with the minimum clock first. For example, in a process with two threads, if Thread 1 is trying to acquire a lock when its logical clock is 1029, it will not be able to do so if Thread 2's clock is at 329, because of being less than 1029. But, as soon as Thread 2's clock get past 1029, Thread 1 will acquire the lock. So basically our purpose is not only to reduce the code that updates the clocks but also to update the clocks as soon as possible, so that logical clock of the thread waiting for a lock becomes minimum more quickly. In fact, at compile time it is possible to increment the clock even before some instructions are executed, since at compile time we can count the number of instructions. Therefore in all optimizations we apply, besides trying to reduce the clock update overhead, we also try to increment the clock as soon as possible. Without any optimization, we update the clock at start of each of the basic block of LLVM IR. If there is a function call inside that block, we split that block, such that each block either contains no function call or starts and end with a function call. Then we update the clock at the top of each block if that block contains no function calls, otherwise we update the clocks in between the function calls. By splitting blocks in such a way, we can more easily apply optimizations.

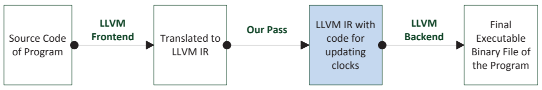
Source Code of Program
LLVM Frontend
Translated to LLVM IR
Our Pass
LLVM IR with code for updating clocks
LLVM Backend
Final Executable Binary File of the Program

Fig. 3. Our tool modifies the LLVM IR code by inserting code for updating logical clocks, used for *deterministic multithreading*

We apply the following optimizations to reduce the logical clock updating overhead as well as reduce waiting time for threads waiting for a lock, by incrementing clocks ahead of time.

Optimization 1 (Function Clocking) As discussed previously, the sooner the clocks are updated, the better, and leaf functions (functions that do not call any function) with only one basic block are perfect candidates for such an optimization. Clocks can be removed from such functions and instead be added to the basic blocks calling such functions. Besides functions with only one blocks, our method also considers leaf functions with multiple blocks, given that there are no loops in such functions. If our pass sees that all possible paths taken by such a function do not differ by much, we calculate the mean value for all possible paths and use that mean value to update the clock. We call such leaf functions as *clocked functions*. By intuition, we can judge that it is also possible to clock functions which call only *clocked functions*. In this way, we can even clock functions which are not necessarily leaf functions.

Optimization 2 (Conditional Blocks Optimization) This optimization is based on the principle that if a block has two or more successors, given that those successors are not merge nodes, we can make the successor with the least clock zero and subtract its original value from all its siblings, while also adding its original clock to the parent block. Another principle of this optimization is that if all predecessors of a merge block have that merge block as their only successor, the clocks could be shifted from the merge node to them. It should be noted that after having parsed all the blocks of a function and applying this optimization, if it is still possible to apply this optimization once more, it is applied.

Optimization 3 (Averaging of Clock) This optimization is based on the fact that paths emanating from a block in a function could be matching close together in total clock values. One can imagine it as a specialized case of *Function Clocking*. For *Function Clocking*, we just considered the paths emanating from the entry block, but here we also check for paths besides the entry block. When forming paths for a block, we only consider blocks dominated by it (execution must pass through the dominating block to reach its dominated blocks). We also make sure there are no loops or unclocked functions in such paths. If we find such a block in a function, we remove clocks from all the blocks in the averaged path and assign the mean clock value to that block.

Optimization 4 (Loops Optimization) This optimization considers the fact that loops are often executed multiple times. So for example, if you have a *for* loop, the increment operation will take place just before the next iteration. Therefore we check for back edges and if we see that the clock of the block from which the backedge is originating is less that a certain threshold value and is also less than the clock of the block it is jumping to, we merge its clock value to that block's clock and remove clock updating code from it.

4) System Calls, Non-deterministic functions and I/O: We use LD_PRELOAD to preload the system call wrappers found in *glibc* with our own version which perform logging of type, input parameters and output of the system calls. Type and input parameters are then read by the follower for error detection, while the output logged by the leader is used directly by the follower, instead of actually executing the system call. Working only with user-space wrappers of system calls is possible, because most of the system calls are usually called through their user-space wrapper functions. This method will not work however, if for example, a system call is made without using the wrapper function, for example, by using inline assembly. So, with our library, the programmer needs to make sure to not make a system call directly. Since the glibc library sometimes also make system calls directly, for example, by making the *clone* system call in *pthread_create* function, we provide our version of *pthread_create*. We also provide our own version of non-deterministic functions such as *rand* and preload them using LD_PRELOAD. Follower uses the output logged by the leader for such functions. Each system call is protected by a deterministic lock to make sure that system calls occur in the same order in the replicas. For the system call *mmap*, which modifies the address space of the process, we take a special approach. The follower still uses the returned addressed by the leader, but use it as a parameter combined with *MAP_FIXED* flag to call the *mmap* function, to ensure that the follower uses the same memory addresses as the leader.

For I/O, our library allows deterministic I/O for sequential file access and screen write. Write to a file or screen is only performed after making sure that no error occurred during an epoch. For that purpose, no output is committed during an epoch. Instead it is buffered. Our library overrides the *write* and *read* system call wrappers to allow buffering of the data. The buffers are committed at the end of an epoch after comparing the buffer contents of the leader and follower by using hash-sums. For this purpose, each file opened for writing is allocated a special buffer. It is important that addresses of these buffers are the same for the leader and follower process. For this purpose, we use a deterministic memory allocation scheme like the one described in Section IV-A2. For sequential file reading, the file offset value is saved at the end of each epoch, so that the file can be rewinded to the previous value in case of rollback.

B. Error detection

At regular intervals of 1 second, known as epochs, dirtied (modified) memory pages of the leader and follower processes are compared. However, the epoch time is reduced to 100 ms if a file or screen output occurs during the epoch. Instead of comparing each memory one by one, the leader and follower processes calculate hash-sums of the dirtied (modified) mem-

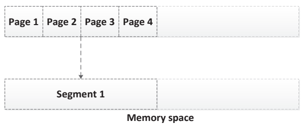

Fig. 4. Memory pages can be grouped into segments to reduce the overhead of memory comparison for error detection

ory pages, which are then compared. If a discrepancy is found, a fault is detected.

The comparison is made even faster by assigning each thread to calculate hash-sum of different portions of the memory. The leader keeps its hash-sums in shared memory so that the follower can read it from there for comparison. We perform memory comparison at barriers which are already found in the program rather than stopping and creating a barrier. This improves the performance, as threads already wait for each other at barriers. Otherwise we create barriers at system call points if necessary (at the end of an epoch that is).

Since our scheme runs at the user level, we cannot note down dirtied pages while handling page faults (from the kernel), the way *Respec* does, which is the most efficient method possible. Therefore, we take special steps to improve its performance. At start of each epoch, we give only read access to allocated memory pages. Whenever a page is accessed for writing, the OS sends a signal to the accessing thread. In the signal handler, the address of the memory page is noted down and both read and write accesses are given to that memory page. In this way, we only need to compare the dirtied memory pages at the end of an epoch. Sending signals on each memory page access violation can slow down execution. Therefore, to reduce the number of such signals, we exploit the concept of spatial locality of data and segmented memory into multiple pages, as shown in Figure 4. A write on any part of a read protected segment of N pages is handled by giving write access to all the N pages in that segment.

Some functions, like that for comparing memories, change the stacks differently for the leader and follower threads. For those purposes, we switch to a temporary stack, so that the original stack remains unaltered from such functions.

The watchdog process is used to detect and recover from timeout errors. Details can be found in [6].

C. Recovery

As discussed previously, for fault recovery, we use checkpoint/rollback. Whenever the leader takes a checkpoint, it kills the previous checkpoint. If the leader process detects an error, or the watchdog process detects a hang, a signal is sent to the last checkpoint process, so that the checkpoint process can start execution. The leader and its follower are killed at that point. The checkpoint process then assumes the role of the leader and forks its own follower. It also creates a new checkpoint. Checkpoints are taken only at barrier points. For creating a multithreaded follower, we have implemented a special *multithreaded fork* function that replicates the leader process to create the follower. More detail on this can be found in [6].

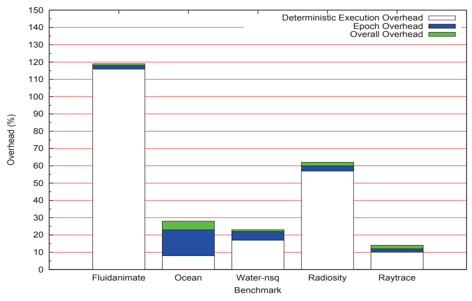

Fig. 5. Overhead of our Fault Tolerance Scheme

V. Performance evaluation

We selected 5 benchmarks, 1 from the PARSEC [2] and 4 from the SPLASH-2 [9] benchmark sets. We ran all our benchmarks on an 8 core (dual socket with 2 quad cores), 2.67 GHz Intel Xeon processor with 32GB of RAM. All programs were compiled with maximum optimization enabled (level -O4 for clang/llvm). All benchmarks were run with 4 threads, meaning we had 8 threads overall for the 2 redundant processes.

A. Results

The results are shown in Table I. In this table, by *Redundant Exec*, we mean results obtained by allowing the leader and follower processes execute freely, without any deterministic execution and fault tolerance, while *Deterministic Exec* overhead is the overhead with deterministic execution only, whereas *Overall Exec* overhead includes all the components of our fault tolerance scheme. For overall execution, the results are shown with memory grouping size of 4. Figure 5 shows the different overheads separately. The epoch overhead here represents overhead of checkpointing, signals for noting dirtied (modified) memory pages and watchdog process. For benchmarks with high lock frequencies (*Fluidanimate* and *Radiosity*, the overhead of deterministic execution is expectedly quite large, whereas for *Ocean* which has high memory usage, epoch overhead is the most. This is because for *Ocean*, large number of signals are received when memory pages are modified during an epoch.

B. Deterministic execution overhead

Figure 6 shows the deterministic execution performance improvement that we get by applying optimizations on the compiler pass that inserts code to update the logical clocks. The lower portion of the bars (in white color) in Figure 6 show the overhead of deterministic execution with the optimizations, while the upper portion (in blue color) show the additional overhead when optimizations are not applied.

We get improvement in performance due to two reasons. Firstly, because of reducing the clock updating code and secondly by updating clocks ahead of time (See Section IV-A3

TABLE I
PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmark	**Fluidanimate**	**Ocean**	**Water-nsq**	**Radiosity**	**Raytrace**
Original Exec Time (ms)	1142	1870	1416	962	677
Locks/sec	6266655	5397	132798	6072689	225635
Pages Compared	3628	40182	371	8730	109
Epochs	3	3	2	2	2
Redundant (Redt) Exec Time and Overhead	1204 (5%)	1872 (0%)	1441 (2%)	980 (2%)	697 (3%)
Deterministic Exec Time and Overhead with No Optimization (w.r.t Redt Exec)	3072 (155%)	2022 (8%)	2102 (46%)	1892 (93%)	813 (17%)
Deterministic Exec Time and Overhead with Optimizations (w.r.t Redt Exec)	2603 (116%)	2014 (8%)	1688 (17%)	1534 (57%)	767 (10%)
Overall Exec Time and Overhead (w.r.t Redt Exec)	2639 (119%)	2391 (28%)	1771 (23%)	1584 (62%)	795 (14%)

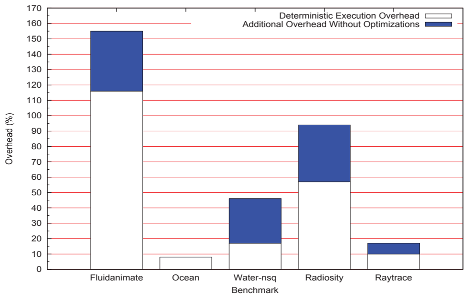

Fig. 6. Improvement in Deterministic Execution Performance by applying optimizations

for discussion on the deterministic algorithm that we use and why updating clocks ahead of time is beneficial). For *Radiosity*, Optimization 1 (Function Clocking) is applicable on a large number of functions, with some of them being quite compute intensive. Therefore, by incrementing the clocks for those clockable functions ahead of time, we significantly reduce the waiting time for threads which are about to acquire a lock. For *Fluidanimate* and *Water-nsq*, the improvement was mostly due to the Optimization 4 (Loops Optimization) because these benchmarks contain compute intensive small loops. Optimization 3 (Averaging of Clock) worked well for *Raytrace* because our compiler pass could find such paths (for whom clocks could be averaged) in it. Furthermore, Optimization 2 (Conditional Blocks Optimization) was useful for reducing the clock overhead of most of the benchmarks, because such conditional paths are commonly found in programs.

VI. CONCLUSION

In this paper, we described the design and implementation of a user-level leader/follower based fault tolerance scheme for multithreaded applications running on multicore processors. Instead of using record/replay technique to ensure deterministic shared memory accesses by the replicas, we used *deterministic multithreading*, where the redundant processes do not need to communicate with each other for ensuring deterministic shared memory accesses. This improves isolation between the redundant processes, increasing fault tolerance and reliability, besides consuming less memory. To increase portability, we avoid using any special hardware for deterministic execution and modifying the kernel. We instead implemented a compiler pass that inserts code to update logical clocks for *deterministic multithreading*. We also applied several optimizations to reduce the overhead of logical clock updating code. In the absence of faults, our fault tolerance scheme, adds an average overhead of 49% for selected benchmarks, with 4 threads.

ACKNOWLEDGMENT

This research has been funded by the projects Smecy 100230, iFEST 100203 and REFLECT 248976.

REFERENCES

[1] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.
[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. PACT '08, pages 72–81.
[3] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. HPCA '11, pages 333 –334, feb.
[4] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, ASPLOS '10, pages 77–90.
[5] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44:97–108, March 2009.
[6] H. Mushtaq, Z. Al-Ars, and K. Bertels. A user-level library for fault tolerance on shared memory multicore systems. In *Proc. 15th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, Tallinn, Estonia, April 2012.
[7] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. SOSP '11, pages 327–336.
[8] V. Weaver and S. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141 –150.
[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995.
[10] R. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, 1(1):17 –22, March 2001.
[11] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. ISCA '11, pages 201–212.
[12] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.
[13] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):155–166, June 2010.
[14] A. Basu, J. Bobba, and M. D. Hill, "Karma: scalable deterministic record-replay," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11, 2011, pp. 359–368.
[15] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007.

5 COMPARISON OF FAULT TOLERANCE METHODS

This chapter compares the two different schemes that we developed for fault tolerance on shared memory multicore systems. The first scheme uses record/replay technique where one replica logs the order of shared memory accesses and the others read from that log, while the second scheme uses deterministic multithreading, meaning that for the same input, a multithreaded program always has the same lock interleaving. Unlike record/replay systems, this eliminates the requirement for communication between the redundant processes for shared memory accesses. Both of our schemes are implemented totally in software, requiring no special hardware, thus making them very portable. Furthermore, our schemes are totally implemented at the user-level, requiring no modification of the kernel. We also implemented a hardware extension to improve the performance and scalability of deterministic multithreading. We compared our deterministic multithreading scheme (DetLock) with the state of the art Kendo. We showed that for several benchmarks, especially those with high frequency of shared memory access, our scheme outperforms Kendo, due to the ability of updating the clocks more frequently and ahead of time.

For record/replay, the overhead is less than 25% for all benchmarks down to approximately 0% for some benchmarks. For deterministic multithreading, it depends on the usage of shared memory. With low shared memory usage, the overhead is less than 25% while with high usage, it can reach up to 56%. We also compare our deterministic multithreading scheme (DetLock) to other approaches, for example Kendo, and show overhead reduction up to 189% for 8 threads. Even when Kendo is assisted with hardware, the hardware assisted DetLock version can improve performance up to 26% over Kendo.

This chapter is based on the following paper.

Mushtaq, H.; Al-Ars, Z.; Bertels, K., *Fault Tolerance on Multicore Processors using Deterministic Execution*, Submitted to IEEE transactions on reliability.

Fault Tolerance on Multicore Processors Using Deterministic Execution

Hamid Mushtaq, Koen Bertels, Zaid Al-Ars
Computer Engineering Laboratory
Delft University of Technology
{H.Mushtaq, K.L.M.Bertels, Z.Al-Ars}@tudelft.nl

***Abstract*—This paper describes and compares two different schemes for fault tolerance on shared memory multicore systems. The first scheme uses record/replay technique where one replica logs the order of shared memory accesses while other replicas read from that log. The second scheme uses deterministic multithreading, meaning that for the same input, a multithreaded program always has the same lock interleaving. Unlike record/replay systems, this eliminates the requirement for communication between the redundant processes for shared memory accesses. Both of our schemes are implemented totally in software, and require no modification of the kernel. For record/replay, the overhead is less than 25% for all benchmarks down to approximately 0% for some benchmarks. For deterministic multithreading, it depends on the usage of shared memory. With low shared memory usage, the overhead is less than 25% while with high usage, it can reach up to 56%. We also implemented a hardware extension to aid in deterministic multithreading, to improve the performance and scalability. We compared our deterministic multithreading scheme (DetLock) to other approaches, for example Kendo, and showed overhead reduction up to 189% for 8 threads. Even when Kendo is assisted with hardware, the hardware assisted DetLock version can improve performance up to 26% over Kendo.**

***Keywords*-multicore, deterministic execution, shared memory, fault tolerance, reliability.**

I. INTRODUCTION

The abundant computational resources available in multicore systems have made it feasible to implement otherwise prohibitively intensive tasks on consumer grade systems. However, these systems integrate billions of transistors to implement multiple cores on a single die, thus raising reliability concerns, as smaller transistors are more susceptible to transient [14] as well as permanent [15] faults.

A common approach for providing fault tolerance is to perform redundant execution of the software. This is done by using the state machine replication approach [16]. In this approach, the replicated copies of a process (known as replicas) follow the same execution sequence and produce the same output if given the same input. This requirement necessitates that the replicas handle non-deterministic events such as asynchronous signals and non-deterministic functions (such as *gettimeofday*) deterministically. This is usually done by having one replica log the non-deterministic events and have the other replicas replay them at the same point in program execution. In a shared memory multithreaded program, this also means that the replicas perform non-deterministic shared memory accesses deterministically, so that they do not diverge in the absence of faults.

One way of making sure that the redundant processes access the shared memory in the same order is to perform record/replay where the leader process records the order of locks (to access shared memory) in a queue which is shared between the leader and follower. The follower in turn reads from that queue to have the same lock acquisitions order. This approach is used by Respec [6] and our previous work [10]. This is the first approach that we have used in this paper. The second approach that we use is deterministic multithreading, where given the same input, a multithreaded process always has the same lock interleaving. This makes sure that the redundant processes acquire the locks in the same order without communicating with each other. We adapt the method used by Kendo [7] to do this, but unlike Kendo, our scheme neither requires deterministic hardware performance counters, which are not available on many platforms [12] (including many x86 systems), nor kernel modification for deterministic execution. The logical clocks used for deterministic execution are inserted by the compiler instead. Moreover, we apply several optimization to further improve the deterministic execution performance. Furthermore, we also implemented hardware extensions to aid in deterministic multithreading. Having hardware decreases portability, but gives significant improvement in performance and scalability.

We can sum up the contributions of this paper as follows.

1) We discuss the implementation of our two schemes for deterministic execution on multicore platforms for fault tolerance. Both of our schemes are implemented using a user-level library and do not require a modified kernel. Secondly, both of our schemes are very portable since they do not depend upon any special hardware for deterministic execution.
2) We compare the advantages and disadvantages of both schemes in terms of performance, memory consumption and reliability.
3) We compare with existing approaches and show that our schemes have better performance on most of the selected benchmarks.
4) We also implement hardware extensions for deterministic multithreading to improve in performance and scalability.

In Section II we discuss the background and related work. In Section III, we give an overview of our fault tolerance method. This is followed by Section IV, where we discuss the detailed fault tolerance implementation. In Section V, we discuss record/replay, while in Section VI, we discuss deterministic multithreading. In Section VII, we evaluate the performance of our method. Finally, we conclude the paper with Section VIII.

II. Background and related work

A fault tolerant system which uses redundant execution needs to make sure that the redundant processes do not diverge in the absence of faults, that is, they should have the same states for the same input. In a single threaded program, in the absence of any fault, the only possible causes of divergence among the replicas can be non-deterministic functions (such as *gettimeofday*) or asynchronous signals/interrupts.

However, in multithreaded programs running on multicore processors, there is one more source of non-determinism, which is shared memory accesses. These accesses are much more frequent than interrupts or signals. Therefore, efficient deterministic execution of replicas in such systems is much more difficult to achieve.

One method to ensure redundant processes access shared memory in the same order is record/replay. In this method, all interleaving of shared memory accesses by different cores or processors are recorded in a log, which can be replayed to have a replica which follows the original execution. Examples of schemes using this method are Rerun [5] and Karma [1]. These schemes intercept cache coherence protocols to record inter-processor data dependencies, so that they can be replayed later on, in the same order. While Rerun only optimizes recording, Karma optimizes both recording and replaying, thus making it suitable for online fault tolerance. It shows good scalability as well. The disadvantage of record/replay approaches as compared to deterministic multithreading is that they require a large memory for recording. Moreover, when used for fault tolerance, the redundant processes need to communicate with each other for shared memory accesses, as one replica records the log while the other reads from it. Respec [6] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rolls-back and re-executes from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed, which has a larger overhead.

The disadvantage of employing record/replay for deterministic shared memory accesses is that it requires communication between the replicas for shared memory accesses, making the fault tolerant method less reliable as the shared buffer used for communication can itself become corrupted by one of the replicas. Moreover it requires extra memory.

To eliminate this communication and memory requirement for shared memory accesses, we can employ deterministic multithreading, where a multithreaded process has always the same memory interleaving for the same input. The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions, that is, provide Strong Determinism. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [2], which has a maximum overhead of 11x for 8 cores. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [11]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. An example of such approach is Calvin [4], which uses the same concept as CoreDet for deterministic execution but make use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, one can relax the requirements to improve efficiency. For example, Kendo [7] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without race conditions. The authors of Kendo call it Weak Determinism. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [20], Weak Determinism is sufficient for most well written multithreaded programs.

The basic idea of Kendo is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, Kendo still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counters that are deterministic [12]. Secondly, Kendo needs modification of the kernel to allow reading from the hardware performance counters.

III. Fault tolerance method

Our fault tolerant method is intended to reduce probability of failures in the presence of transient faults. The block diagram of our fault tolerance method is shown in Figure 1.

Initially, the leader process (which is the original process highlighted in the figure) creates the watchdog and follower processes. The follower process is identical to the leader process and follows the same execution path. The execution is divided into time slices known as epochs. An epoch starts and ends at a program barrier. At the end of each epoch, the memories of the leader and follower processes are compared. If no divergence is found, a checkpoint is taken and the output to files or screen is committed. The previous checkpoint is also deleted. The checkpoint is basically a suspended process which is identical to the leader process at the time the checkpoint is taken. If a divergence is found at the end of an epoch, execution is restarted from the last checkpoint by resuming

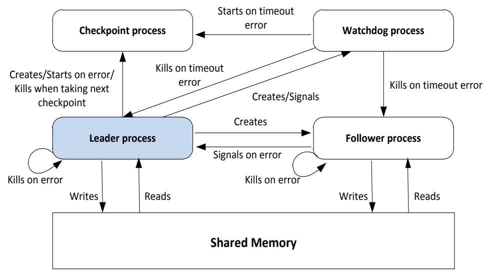

Fig. 1. Block diagram of our fault tolerance method

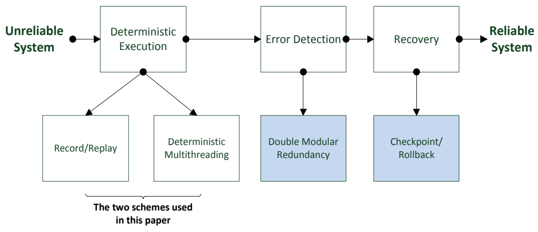

Fig. 2. Stages of our fault tolerance method

the checkpoint process and killing the leader and follower processes. When the checkpoint process starts, it becomes the leader and creates its own follower. It might also happen that the leader or follower processes are unable to reach the end of an epoch, due to some error which hangs them. In that case, the watchdog process detects those hangs by using timeouts and signals the checkpoint process to start. The watchdog process itself is less vulnerable to transient faults as it remains idle most of the time.

Figure 2 shows the stages of fault tolerance that our method implement. The method starts by insuring deterministic execution, followed by error detection and then recovery. For deterministic execution, we implemented and evaluated two different schemes, record/replay (See Section V) and deterministic multithreading (See Section VI).

IV. IMPLEMENTATION

In this section, we discuss the implementation of our fault tolerance method. We start by Section IV-A, where we discuss how deterministic execution of the replicas is performed. This is followed by Section IV-B which discusses error detection. Finally in Section IV-C, we discuss our recovery mechanism.

A. Deterministic execution

For deterministic execution, we need to ensure that replicas use the same memory addresses. We also need to ensure determinism in the presence of non-deterministic functions and shared memory accesses. Moreover, we need to make sure that the leader and follower processes use the same memory addresses. For this we need to have a deterministic memory allocation scheme. Finally we also need to make sure that we have deterministic I/O. Below we discuss how we handle these issues.

1) Replica creation: Our library assumes that threads in the application are created once at the start of the application. Therefore, we create the follower process at that point in the code where the threads are created. For this purpose, we replace the *pthread_create* function with our own to make sure the threads of the replicas use the same memory addresses. More detail on this can be found in [10].

2) Memory allocation: We implement our own memory allocation functions to allocate memory deterministically. Our implementation replaces the locks in the original memory allocation functions with our own deterministic locks. The variables used by our library (not related to original program execution) to perform deterministic execution, may have different values for the leader and follower processes, for example, the flag used to distinguish the leader process from the follower process. For these variables, we use a separate memory, which is allocated with the *mmap* system call. This memory is not compared for error detection.

3) Deterministic shared memory accesses: We have used two different scheme for ensuring deterministic shared memory accesses, as shown in the Figure 2. The first one is record/replay, discussed in Section V and the second one is deterministic multithreading, discussed in Section VI

4) System calls, non-deterministic functions and I/O: We use LD_PRELOAD to preload the system call wrappers found in *glibc* with our own version which performs logging of type, input parameters and output of the system calls. Type and input parameters are then read by the follower for error detection, while the output logged by the leader is used directly by the follower, instead of actually executing the system call. Working only with user-space wrappers of system calls is possible, because most of the system calls are usually called through their user-space wrapper functions. This method will not work however, if a system call is made without using the wrapper function, for example, by using inline assembly. So, with our library, the programmer needs to make sure not to make a system call directly. Since the *glibc* library sometimes also make system calls directly, for example, by making the *clone* system call in *pthread_create* function, we provide our version of *pthread_create*. We also provide our own version of non-deterministic functions such as *rand* and preload them using LD_PRELOAD. Follower uses the output logged by the leader for such functions. Each system call is protected by a deterministic lock to make sure that system calls occur in the same order in the replicas. For the system call *mmap*, which modifies the address space of the process, we take a special approach. The follower still uses the returned addressed by the leader, but use it as a parameter combined with *MAP_FIXED* flag to call the *mmap* function, to ensure that the follower uses the same memory addresses as the leader.

For I/O, our library allows deterministic I/O for sequential file access and screen write. Write to a file or screen is only performed after making sure that no error occurred during an epoch. For that purpose, no output is committed during an epoch. Instead it is buffered. Our library overrides the

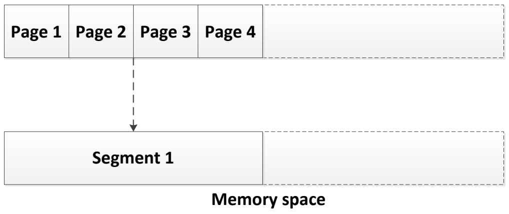

Fig. 3. Segmentation of memory to reduce memory checking overhead

write and *read* system call wrappers to allow buffering of the data. The buffers are committed at the end of an epoch after comparing the buffer contents of the leader and follower by using hash-sums. For this purpose, each file opened for writing is allocated a special buffer. It is important that addresses of these buffers are the same for the leader and follower process. For this purpose, we use a deterministic memory allocation scheme like the one described in Section IV-A2. For sequential file reading, the file offset value is saved at the end of each epoch, so that the file can be rewinded to the previous value in case of rollback.

At this moment, our fault tolerance scheme does not work with programs that use inter process communication (through pipes and shared memory for example). The only form of I/O allowed is disk I/O and screen output. Moreover, our scheme assumes that there are no data races in the program. Lastly, we have not added functionality to handle asynchronous signals. However, this functionality can be added for user space by handling asynchronous signals at synchronous points, such as system calls, as done by Scribe [17].

B. Error detection

At regular intervals of 1 second, known as epochs, dirtied (modified) memory pages of the leader and follower are compared. However, the epoch time is reduced to 100 ms if a file or screen output occurs during the epoch. Instead of comparing each memory one by one, the leader and follower processes calculate hash-sums of the dirtied (modified) memory pages, which are then compared. If a discrepancy is found, a fault is detected.

The comparison is made even faster by assigning each thread to calculate hash-sum of different portions of the memory. The leader keeps its hash-sums in shared memory so that the follower can read it from there for comparison. We perform memory comparison at barriers which are already found in the program rather than stopping and creating a barrier. This improves the performance, as threads already wait for each other at barriers. If insufficient barriers are found in the program, the programmer can insert calls to function *potential_barrier_wait*, which is provided by our library. This function creates a barrier only when required, that is at the end of an epoch.

Since our scheme runs at the user level, we cannot note down dirtied pages while handling page faults (from the kernel), the way Respec does. Therefore, we take special steps to improve its performance. At start of each epoch, we give only read access to allocated memory pages. Whenever a page is accessed for writing, the OS sends a signal to the accessing thread. In the signal handler, the address of the memory page is noted down and both read and write accesses are given to that memory page. In this way, we only need to compare the dirtied memory pages at the end of an epoch. Sending signals on each memory page access violation can slow down execution. Therefore, to reduce the number of such signals, we exploit the concept of spatial locality of data by segmenting memory into multiple pages, as shown in Figure 3. A write on any part of a read protected segment of N pages is handled by giving write access to all the N pages in that segment.

Some functions, like those for comparing memories, change the stacks differently for the leader and follower threads. For those purposes, we switch to a temporary stack, so that the original stack remains unaltered by such functions.

The watchdog process is used to detect and recover from timeout errors. Details can be found in [10].

C. Recovery

As discussed previously, for fault recovery, we use checkpoint/rollback. Whenever the leader takes a checkpoint, it kills the previous checkpoint. If the leader process detects an error, or the watchdog process detects a hang, a signal is sent to the last checkpoint process, so that the checkpoint process can start execution. The leader and its follower are killed at that point. The checkpoint process then assumes the role of the leader and forks its own follower. It also creates a new checkpoint. Checkpoints are taken only at barrier points. For creating a multithreaded follower, we have implemented a special *multithreaded fork* function that replicates the leader process to create the follower. More detail on this can be found in [10].

V. Record/replay

For redundant deterministic execution, it is necessary that the leader and follower processes perform shared memory accesses in the same order. For this purpose, a mutex is enclosed in a special data structure, which also contains a pointer to clocks for that mutex to aid in deterministic execution. Whenever a thread in the leader process acquires a mutex, it increments the mutex's clock. A thread in the follower only acquires the same mutex in its execution, when its clock matches that for the corresponding thread in the leader.

We create our own deterministic versions of pthread's synchronization functions Since *pthread_mutex_lock* is the most widely used and is also used in our implementation of other pthread synchronization functions, we discuss our *pthread_mutex_lock* algorithm here, which is shown in Algorithm 1. We also have our own versions of data structures for representing the synchronization objects, for example, *pthread_mutex_log_t* instead of *pthread_mutex_t*. Here *m* represents an object of *pthread_mutex_log_t* structure which holds a mutex and its clocks. There is one such object for each mutex in the program. Therefore, deterministic access to

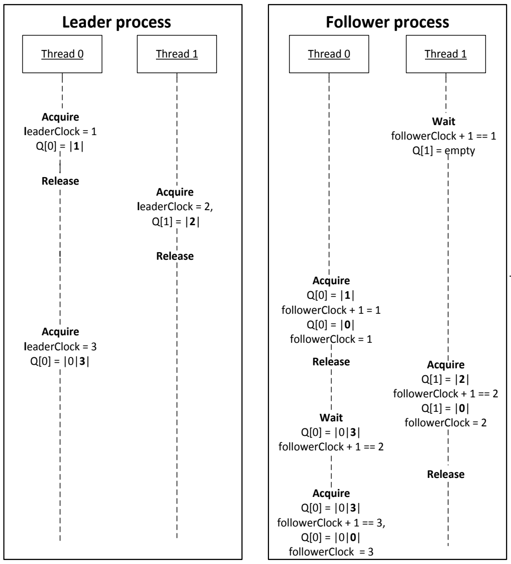

Fig. 4. An example sequence for deterministic shared memory access with a mutex, using our algorithm

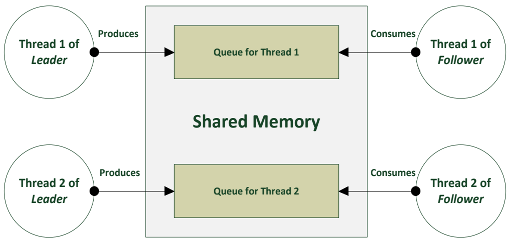

Fig. 5. Communication between the leader and follower processes for deterministic execution

a mutex is independent of other mutexes in the program, hence improving scalability.

When a leader thread acquires a mutex, it increments the leader's clock for that mutex and also records that value in a circular queue, so that the follower can acquire the thread when its clock reaches one less than the same value. The communication between the leader and follower processes is shown in Figure 5. After acquiring the mutex, the follower also increments its clock for that mutex. An example sequence using our deterministic lock and unlocking algorithm is shown in Figure 4, where we can see that threads in the follower process acquire a mutex in the same order as the leader process. The queue elements which the threads are currently writing to or reading from are shown in bold. We can see that the thread 1 of the follower process is unable to acquire the mutex when the queue is empty. Also thread 0 of the follower process is unable to acquire the mutex until thread

Algorithm 1 Pseudocode for deterministic lock

```
function R_PTHREAD_MUTEX_LOCK(ref pthread_mutex_log_t m)
    q = GetQueue(tid)                              ▷ There is a separate queue for each thread
    if isLeader then
        r = lock(m.mutex)
        if r == 0 then                       ▷ Only if lock call is successful, increment the clock
            m.clock = m.clock + 1                       ▷ m.clock does not need to be atomic
        end if
        while !pushq(q, MUTEXLOCK, m.mutex, m.clock, r)
        end while
        return r
    else                                                                       ▷ Follower
        while not !popq(q, ref type, ref mutex, ref clock, ref r)
        end while
        if type != MUTEXLOCK and mutex != m.mutex then    ▷ Logged parameters do not match
            SignalErrorAndExit()
        end if
        if r != 0 then
            return r
        end if
        while (m.clock+1) != clock
        end while
        lock(m.mutex)
        m.clock = m.clock + 1
        return 0
    end if
end function

function PUSHQ(q, type, addr, clock, r)                                ▷ Called by Leader
    lindex = GetLeaderQIndex(tid)
    if checkQElementsForZero(lindex) then
        q[lindex].type = type
        q[lindex].addr = addr
        q[lindex].clock = clock
        q[lindex].r = r + 1
        SetLeaderQIndex((lindex + 1) %QCAPACITY)
        return TRUE
    else
        return FALSE
    end if
end function

function POPQ(q, ref type, ref addr, ref clock, ref r)               ▷ Called by Follower
    findex = GetFollowerQIndex(tid)
    if checkQElementsForNonZero(qindex) then
        type = q[findex].type
        addr = q[findex].addr
        clock = q[findex].clock
        r = q[findex].r - 1
        setQElementsToZero(findex)
        SetFollowerQIndex((findex + 1) %QCAPACITY)
        return TRUE
    else
        return FALSE
    end if
end function
```

0 increments *followerClock* such that *followerClock+1* equals the value which is being read from the queue.

Unlike Respec which uses a hash table of 512 entries to keep clocks for all the synchronization objects, we use a separate clock for each mutex. The benefit of this is that we can avoid using atomic variables for accessing the clocks, as the clock can be incremented after acquiring the lock. We also optimize the queue access by avoiding using atomic variables and avoiding true and false sharing of cache lines. For that purpose, we use a lockless queue as shown by *pushq* and *popq* functions in Algorithm 1. This is unlike Respec which uses atomic operations if necessary to access the queue. The typical method of using a lockless queue (which we call *naive* in this paper) is to use shared tail and head indexes. Since in this method, producer and consumer read the head or tail indexes at the same time when the other is writing to it, this causes cache trashing. Hence it is a *true sharing* problem. We avoid this by having local indexes for producer (leader) and consumer (follower). The check for emptiness and fullness is done by checking the data value instead. Producer only writes to the queue when all the queue element it is about to write to, is zero, while the consumer only reads when the queue element is non-zero. Here, since the value of *r*, which represents the result returned by a synchronization function can be zero, we add one to its value while pushing and subtract one from it when popping. We make sure that the indexes for leader and follower do not share the same cache line by having sufficient padding between them. This makes sure that we do not have the problem of *false sharing*.

VI. Deterministic Multithreading

Our tool for deterministic multithreading which we call DetLock, uses Kendo's algorithm to perform deterministic execution. However, unlike Kendo which requires deterministic hardware performance counters, which are not available on many platforms, such as most Intel and AMD processors. DetLock inserts code to update logical clocks at compile time. This also means that we do not need to modify the kernel which is required by Kendo to read from performance counters. Figure 6 shows the compilation step where our compiler pass executes, which is after generating the LLVM IR (Intermediate Representation) code and before generating the final binary code.

The logical clock is incremented by 1 with each instruction in the code. For instructions which take more than one clock cycles, the logical clock is updated according to the approximate number of clock cycles they take.

The Kendo's method of acquiring locks deterministically works by giving the lock to the thread with the minimum clock first. So basically our purpose is not only to reduce the code that updates the clocks but also to update the clocks as soon as possible, so that the thread waiting for a lock will be able to acquire it more quickly. In fact, at compile time it is possible to combine clock increments together and to increment the clock even before some instructions are executed, since at compile time we can count the number of instructions. Therefore in all optimizations we apply, besides trying to reduce the clock update overhead, we also try to increment the clock as soon as possible. Without any optimization, we update the clock at start of each of the basic block of LLVM IR. If there is a function call inside that block, we split that block, such that each block either contains no function call or starts and ends with a function call. Then we update the clock at the top of each block if that block contains no function calls, otherwise we update the clocks in between the function calls. By splitting blocks in such a way, we can more easily apply optimizations.

We apply the following optimizations to reduce the logical clock updating overhead as well as reduce waiting time for threads waiting for a lock, by incrementing clocks ahead of time. More details about these optimizations can be found in [8] and [9].

A. Optimizations

We apply two type of optimizations. Firstly, the compile time optimizations, which are done during the source code compilation, and secondly, the application level optimizations, that require modifying the source code. These optimizations are discussed below.

1) Compile time optimizations: At compile time, we modify the LLVM intermediate code to improve deterministic execution performance. These optimizations are discussed below.

Optimization 1 (Function Clocking) - As discussed previously, the sooner the clocks are updated, the better, and leaf functions (functions that do not call any function) with only one basic block are perfect candidates for such an optimization. Clocks can be removed from such functions and instead be added to the basic blocks calling such functions. Beside functions with only one blocks, our method also considers leaf functions with multiple blocks, given that there are no loops in such functions. If our pass sees that all possible paths taken by such a function do not differ by much, we calculate the mean value for all possible paths and use that mean value to update the clock. We call such leaf functions as *clocked functions*. By intuition, we can judge that it is also possible to clock functions which call only *clocked functions*. In this way, we can even clock functions which are not necessarily leaf functions.

Optimization 2 - (Conditional Blocks Optimization) This optimization is based on the principle that if a block has two or more successors, given that those successors are not merge nodes, we can make the successor with the least clock zero and subtract its original value from all its siblings, while also adding its original clock to the parent block. Another principle of this optimization is that if all predecessors of a merge block have that merge block as their only successor, the clocks could be shifted from the merge node to them. It should be noted that after having parsed all the blocks of a function and applying this optimization, if it is still possible to apply this optimization once more, it is applied.

There are several more optimizations applied, which are reported in [9].

Optimization 3 - (Averaging of Clock) This optimization is based on the fact that paths emanating from a block in a function could be matching close together in total clock values. One can imagine it as a specialized case of *Function Clocking*. For *Function Clocking*, we just considered the paths emanating from the entry block, but here we also check for paths besides the entry block. When forming paths for a block, we only consider blocks dominated by it (execution must pass through the dominating block to reach its dominated blocks). We also make sure there are no loops or unclocked functions in such paths. If we find such a block in a function, we remove clocks from all the blocks in the averaged path and assign the mean clock value to that block.

Optimization 4 - (Loop Optimization) This optimization considers the fact that loops are often executed multiple times. So for example, if you have a *for* loop, the increment operation will take place just before the next iteration. Therefore we check for backedges and if we see that the clock of the block from which the backedge is originating is less that a certain threshold value and is also less than the clock of the block it is jumping to, we merge its clock value to that block's clock and remove clock updating code from it.

Another part of this optimization is incrementing the clock even before the loop is executed. This is possible if the number

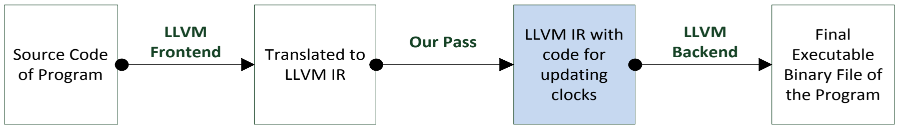
Source Code of Program
LLVM Frontend
Translated to LLVM IR
Our Pass
LLVM IR with code for updating clocks
LLVM Backend
Final Executable Binary File of the Program

Fig. 6. Our tool modifies the LLVM IR code by inserting code for updating logical clocks, used for deterministic multithreading

```
for(i = 0; i < n; i++)
{
  .....
  if(get_lock_cond(i))
  {
    lock(mutex);
    perform_calc( param );
    unlock(mutex);
  }
  else
    perform_calc( param );
}
```

Fig. 7. Example loop for Optimization 5

of iterations are contained in a variable or represented by a number and are therefore known at compile time, and there is no function call inside the loop. If there are conditions inside the loop, the minimum clock of all possible execution paths multiplied by the number of iterations could be updated before executing the loop, and the rest could be updated during the loop execution, but if there are no conditions inside the loop, the number of instructions inside the loop could just be multiplied with the number of iterations of the loop. In this case, there would be no need to update the clock inside the loop. More details about this optimization can be found in [9]

Updating the clock before executing the loop has two benefits. First it reduces the clock updating overhead. Secondly, it also improves deterministic execution time, as now threads can update their clocks faster, and therefore any thread which is waiting for the other threads to go past it in clock values, would have to wait less to acquire the lock.

2) Application level optimizations: Some optimizations are too complex or impossible for the compiler to perform, as they may require the application knowledge to work. Such kind of optimizations require the programmer to slightly modify the code. We discuss these optimizations below.

Optimization 5 - (Loop peeling)

By loop peeling, we can accentuate the effect of certain optimizations that we apply for improving the performance of deterministic execution. For example, with Optimization 4, we can increment the clock even before the loop is executed. However, if we have locks inside the loop, we cannot apply this optimization. By rewriting the loop such that we can have a loop without locks, we can improve the performance.

An example loop is given in Figure 7. Due to mutex lock and unlock calls, we cannot update the clock before the loop is executed. But if we rewrite this loop as shown in Figure 8, the second loop, which starts on line 11, does not contain any

```
 1 flag = 0;
 2 cutoff = 0;
 3 for(i = 0; i < n; ++j)
 4 {
 5   .....
 6   insert_iter_param( param );
 7   flag = flag | get_lock_cond(i);
 8   cutoff = cutoff + !flag;
 9 }
10
11 for(i = 0; i < cutoff; i++)
12 {
13   .....
14   perform_calc(get_iter_param());
15 }
16
17 for(i = cutoff; i < n; i++)
18 {
19   .....
20   if(get_lock_cond(i))
21   {
22     lock(mutex);
23     perform_calc(get_iter_param());
24     unlock(mutex);
25   }
26   else
27     perform_calc(get_iter_param());
28 }
```

Fig. 8. Modified example loop for Optimization 5

locks, and we could update the clock before executing it. Note that it is possible that the condition for mutex lock and unlock is never true during the original loop execution. In that case, the third loop, which starts on line 17, would not be executed at all. However, if the condition does hold true at least once during loop execution, then we note down the first index at which it was true (by using variables *flag* and *cutoff*), and only use the third loop for those indexes.

Optimization 6 - (Mutually exclusive accesses of synchronization objects)

This optimization consists of two parts, which are discussed below.

Optimization 6.1 In some parallel applications, its possible that not all the threads access the same shared memory. For example, in Figure 9, we show a grid based application, where the access to the elements on borders need to be synchronized. These border areas are denoted by Ix (I1, I2 etc). It is obvious that some threads do not share borders with each other. For example, Thread 0 does not share any border with threads 4, 5, 6 and 7. This means that they never access any shared memory between them. Therefore, when comparing for clocks,

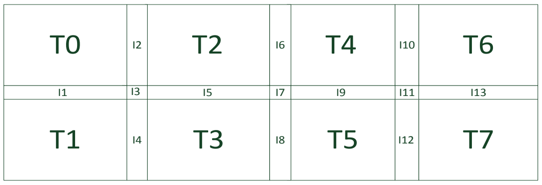

Fig. 9. Mutually exclusivity of shared memory accesses

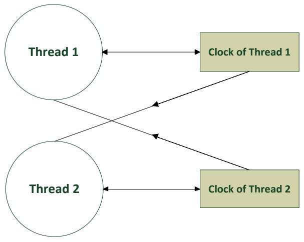

Fig. 10. Communication between threads with software-only deterministic execution

for Thread 0, we do not need to compare its clocks with those of threads 4, 5, 6, 7. In other words, Thread 0 never needs to wait for threads 4, 5, 6 and 7 for acquiring a lock.

Optimization 6.2 Secondly, even for those threads that do share borders, its not always necessary for one thread to wait for another. For example, although threads 0 and 1 share borders, we know for example that when Thread 0 is accessing a shared memory in the region I2, it does not need to wait for Thread 1. This is because Thread 1 would never access any shared memory from the region I2.

B. Hardware extension

Figure 10 shows how the threads update and read clocks for deterministic execution in DetLock. Note that in case of Kendo, the clocks are updated by the kernel on receiving hardware interrupts when a certain store count is reached. For smaller number of cores, the overhead of reading clocks of all threads may not be high. However, this would not scale well with higher number of cores. Moreover, for DetLock, there is also overhead induced by cache coherence, since while one thread is updating its clock, other threads might be reading from it, thus causing invalidations for cache coherence.

For reducing these overheads, we can build a hardware which reads clocks from all the threads, and outputs 1 if a thread satisfies the condition of acquiring a lock and 0 otherwise. In this way, threads do not need to read clocks of the other threads, thus making this process faster, and not causing cache invalidations for maintaining cache coherence. Figure 11 shows the hardware-assisted process of writing the clocks and reading the conditions for acquiring a lock.

We use the same hardware for both the hardware-assisted DetLock and Kendo versions, which are discussed below.

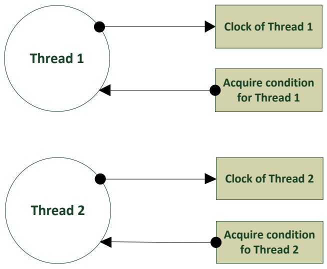

Fig. 11. Communication between threads with hardware-assisted deterministic execution

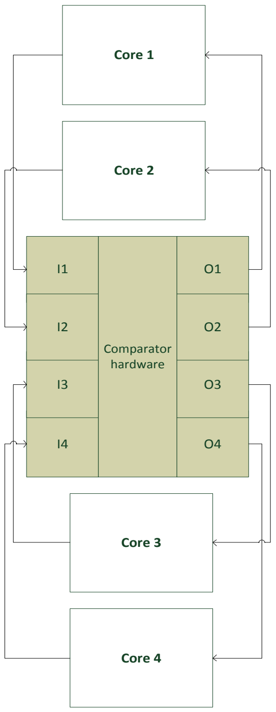

Fig. 12. Block diagram of the proposed hardware to perform deterministic execution

1) DetlockHW: The block diagram of the hardware which assists DetLock is shown in Figure 12. Each core writes to one of the input registers of the comparator. The comparator writes 1 to the output register whose corresponding input register contains the smallest clock value, while writing 0 to all the other output registers. In this way, a thread can know, whether to acquire a lock by just reading the value of its corresponding output register. In case of two or more input registers having the smallest value, the one with the least index has 1 written to its corresponding output register. Through this hardware,

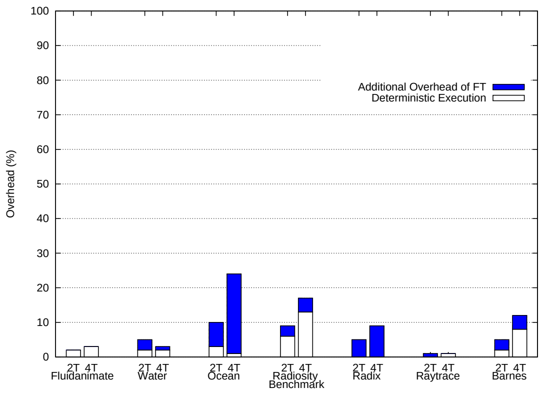

Fig. 13. Performance of record/replay with 2 and 4 threads (2T and 4T)

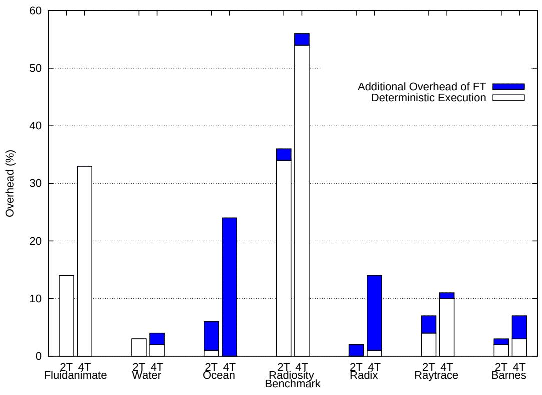

Fig. 14. Performance of deterministic multithreading with 2 and 4 threads (2T and 4T)

there is no need for threads to read other's clocks and also no overhead is incurred due to the cache invalidations that occur for maintaining cache coherence which occurs in DetLock.

For Optimization 6, we can ignore comparing the clock for some cores. Therefore, each core has a register to represent its interaction with another cores, where the bits of that register represent the cores. The clock of a core is compared if its corresponding bit value is 1 and ignored otherwise.

2) KendoHW: The hardware assisted Kendo version uses the same hardware as the hardware assisted DetLock version, except that rather than the program writing clock values to the hardware's input, input is directly taken from hardware performance counters for retired stores. This eliminates the need for programming interrupts that update the clock value, as in case of the original Kendo. This also means that the clock value is updated after each retired store, instead of getting updated after a certain number of stores, which depend upon the chosen chunk size, as in case of the original Kendo version.

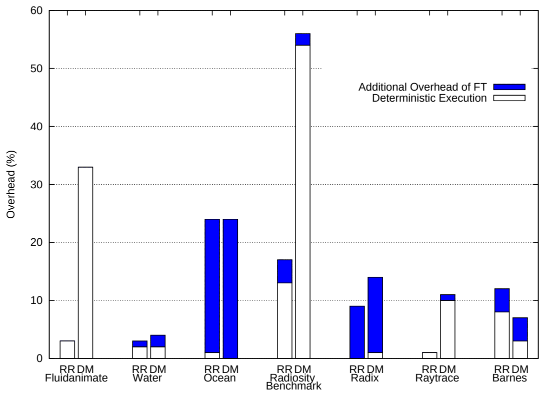

Fig. 15. Comparison of record/replay and deterministic multithreading for 4 threads. Left column is for record/replay (RR) and the right column for deterministic multithreading (DM).

VII. PERFORMANCE EVALUATION

We selected 7 benchmarks, one from the PARSEC [3] and six from the SPLASH-2 [13] benchmark sets. We ran all our benchmarks on an 8 core (dual socket with 2 quad cores), 2.67 GHz Intel Xeon processor with 32GB of RAM. All programs were compiled using the clang compiler with maximum optimization level set. The results are shown in Table I.

In this section, first we compare the performance of record/replay with deterministic multithreading. This is followed by a section which discusses the effect of queue size on the performance of record/replay. Next we compare our record/replay with Respec and our deterministic multithreading scheme with Kendo.

A. Comparison of record/replay and deterministic multithreading performance

The comparison of the performance of record/replay and deterministic multithreading is shown in Table I. For deterministic multithreading, we have shown results both with and without optimizations. The first column contains the benchmark names, while the second column shows the number of threads used. This is followed by the number of mutexes used in the benchmark. Next, locks acquired during the execution are shown. This is followed by the number of memory pages compared for error detection. This is followed by the redundant execution time, which is the time to execute two replicas of the benchmark programs freely, that is, without fault tolerance or deterministic execution. Then we show the deterministic execution time for record/replay, followed by the fault tolerant execution time with record/replay. After that, we show the deterministic execution time with deterministic multithreading, for both unoptimized and optimized versions. Lastly, we show the fault tolerant execution time with deterministic multithreading using the optimized version.

The performance is also illustrated in Figures 13, 14 and 15. Figure 13, shows the overhead for record/replay with two and four threads, while Figure 14 shows the same for deterministic

TABLE I
PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmarks	Threads	Mutexes	Locks	Pages compared	Red exec time (ms)	RR det (ms)	RR FT (ms)	DM unopt (ms)	DM opt (ms)	DM FT (ms)
Fluidanimate	2	73802	3694640	3107	2079	2130 (2%)	2130 (2%)	2583 (24%)	2370 (14%)	2370 (14%)
	4	108242	7156250	2060	1250	1292 (3%)	1295 (4%)	2218 (77%)	1662 (33%)	1667 (33%)
Water	2	527	125047	351	2790	2841 (2%)	2926 (5%)	4462 (60%)	2870 (3%)	2879 (3%)
	4	527	188042	284	1407	1438 (2%)	1449 (3%)	2062 (47%)	1432 (2%)	1465 (4%)
Ocean	2	8	498	299026	5290	5460 (3%)	5844 (10%)	5330 (1%)	5320 (1%)	5572 (5%)
	4	8	996	290633	2898	2917 (1%)	3593 (24%)	2915 (1%)	2899 (0%)	3607 (24%)
Radiosity	2	3918	9001121	29318	1539	1632 (6%)	1680 (9%)	2437 (58%)	2032 (32%)	2066 (34%)
	4	3922	8747666	29340	1005	1136 (13%)	1180 (17%)	1907 (90%)	1544 (54%)	1569 (56%)
Radix	2	78	29	29318	965	966 (0%)	1018 (5%)	968 (0%)	968 (0%)	985 (2%)
	4	86	71	29340	606	609 (0%)	661 (9%)	615 (1%)	615 (1%)	692 (14%)
Raytrace	2	3918	121957	8772	1158	1162 (0%)	1173 (1%)	2606 (125%)	1207 (4%)	1242 (7%)
	4	3922	121959	8950	690	694 (1%)	696 (1%)	816 (18%)	760 (10%)	765 (11%)
Barnes	2	2053	1606025	7624	1652	1678 (2%)	1734 (5%)	1985 (20%)	1684 (2%)	1696 (3%)
	4	2053	1606033	7097	1029	1108 (8%)	1149 (12%)	1126 (9%)	1062 (3%)	1099 (7%)

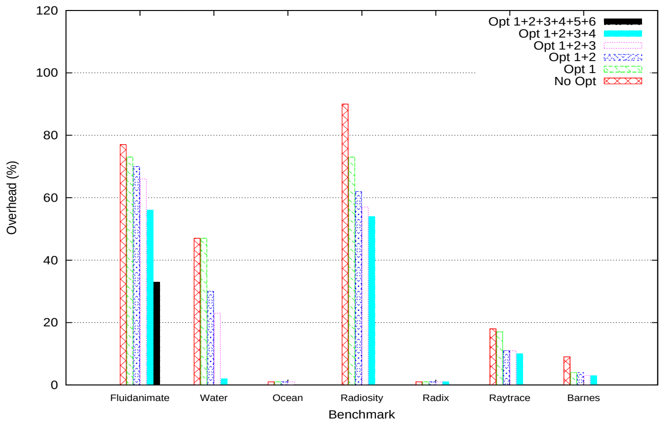

Fig. 16. Reduction in deterministic execution overhead of deterministic multithreading by applying optimizations

TABLE II
PERFORMANCE RESULTS OF OUR SCHEME FOR THE SELECTED BENCHMARKS

Benchmarks	Original	No Opt (ms)	Opt 1 (ms)	Opt 1+2 (ms)	Opt 1+2+3 (ms)	Opt 1+2+3+4 (ms)	Opt 1+2+3+4+5+6 (ms)
Fluidanimate	1250	2218 (77%)	2163 (73%)	2125 (70%)	2075 (66%)	1950 (56%)	1662 (33%)
Water	1407	2062 (47%)	2068 (47%)	1829 (30%)	1736 (23%)	1432 (2%)	-
Ocean	2898	2915 (1%)	2917 (1%)	2913 (1%)	2915 (1%)	2899 (0%)	-
Radiosity	1005	1907 (90%)	1742 (73%)	1620 (62%)	1577 (57%)	1544 (54%)	-
Radix	606	615 (1%)	614 (1%)	616 (1%)	612 (1%)	615 (1%)	-
Raytrace	690	816 (18%)	807 (17%)	765 (11%)	765 (11%)	760 (10%)	-
Barnes	1029	1126 (9%)	1074 (4%)	1072 (4%)	1059 (3%)	1062 (3%)	-

multithreading. Lastly, Figure 15 shows the performance of record/replay side by side with deterministic multithreading for four threads. In all these figures, the lower portion of the bar shows the overhead of deterministic execution, while the whole bar shows the overhead of fault tolerant execution.

The improvement in performance of deterministic multithreading by applying different optimizations is shown in Figure 16. For each benchmark, the leftmost bar shows the overhead without any optimization. The next bar shows the overhead by only applying Optimization 1, this is followed by the bar which shows the overhead with Optimizations 1 and 2 combined. Similarly, the next bar shows the overhead with Optimizations 1, 2 and 3 combined. The right most bar shows the overhead after applying all the optimizations. Since optimizations 5 and 6 was only applied to the Fluidanimate benchmark, for all the other benchmarks, the optimizations are shown only up till Optimization 4. These overheads are also shown in the Table II.

From Figure 15, we can see that record/replay performs better than deterministic multithreading for benchmarks with large shared memory accesses, such as Fluidanimate and Radiosity, but for other benchmarks, there is not much difference. Therefore, in benchmarks with low to moderate shared memory accesses, deterministic multithreading would be a more favorable approach due to its lower memory consumption and less inter-process communication. The scalability of these two approaches also follow a similar trend to their performance, that is, benchmarks with high shared memory accesses scale

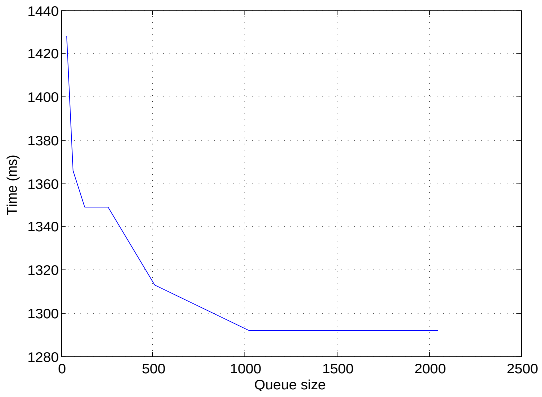

Fig. 17. Effect of queue size on performance of the Fluidanimate benchmark

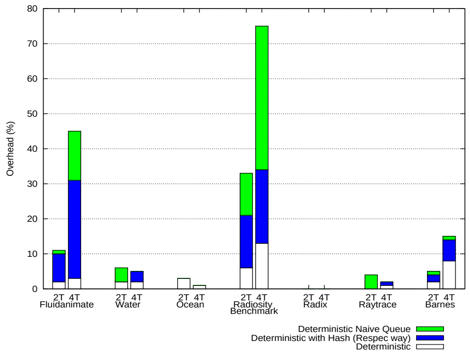

Fig. 18. Comparison of our record/replay approach with that of Respec

better with record/replay, while for other benchmarks there is not much difference.

In Section VI, we discussed how modifying the code can improve the performance for deterministic multithreading. We discussed two optimizations for that purpose, Optimization 5 and 6. We show results for both 4 and 8 threads. The results for the Fluidanimate benchmark are shown in Table III. Here, we give the example of the Fluidanimate benchmark, which had an overhead of 92% with 4 threads and 247% for 8 threads for deterministic execution, even with all optimizations applied. By applying Optimization 5, we reduced this overhead to 57% for 4 threads and 190% for 8 threads, as shown in the fifth column of Table III. Modifying the code also improved the performance without deterministic execution, but even relative to that version, the deterministic execution overhead is just 70% (shown after / in the table) as opposed to 92% for 4 threads and 228% as compared to 247% for 8 threads. We get even further improvement by applying Optimization 6. With that optimization, we were able to reduce the overhead to as much as 35% for 4 threads and 99% for 8 threads.

The Radiosity and Raytrace benchmarks were not race free, and we had to add some more locks to make them race free. However, we noted that for Radiosity, the deterministic multithreading version even performed deterministically if some races were left, indicating that deterministic multithreading provides stronger determinism than record/replay.

B. *Effect of queue size on record/replay performance*

The effect of queue size on record/replay performance of the Fluidanimate benchmark is shown in Figure 17. We can see that the performance increases with the queue size. However, the performance remains the same for queue sizes above 1024. Therefore, the queue size of 1024 is optimal in terms of performance and memory efficiency. Note that for other benchmarks, the queue size did not matter so much. Even with a queue size of 16, they performed as well as with higher queue sizes. The reason queue size matters more in Fluidanimate is because it contains larger number of mutexes, and therefore there is less contention among the threads for the same mutexes.

C. *Comparison with Respec*

Figure 18 shows the improvement that we get by avoiding atomic variables and having an optimized queue. The left bars are obtained by running the benchmarks with 2 threads while right bars are obtained with 4 threads. The lower portion of the bar shows the overhead with our record/replay scheme, where we use a lockless queue and an optimized method for storing the clocks for mutexes. The middle portion shows the additional overhead that Respec needs using a hash table for mutex clocks. The upper most portion shows the additional overhead by using a naive lockless queue.

We can see that for Fluidanimate and Radiosity, which have high lock frequencies, we have a significant improvement in performance of deterministic execution. Furthermore, our method of using separate clocks for each mutex is more scalable than Respec's method of using limited clocks and accessing them through a hash table, that requires atomic operations. The scalability here can be assessed by the fact that for two threads, our scheme and Respec's scheme perform similarly, while for four threads, our scheme performs far better. From this result, we can predict that our scheme will have even better results compared to Respec for larger number of cores. Furthermore, our lockless queue also shows much better scalability than a naive lockless queue due to avoiding true and false sharing of cache lines.

D. *Comparison of DetLock with Kendo*

In Table IV, we compare DetLock with Kendo. Note that the purpose of our scheme is not to surpass Kendo in performance but to make it more portable while retaining sufficient efficiency. We also compare the hardware assisted versions of DetLock and Kendo in this table. For performance evaluation, we selected two benchmarks from PARSEC, which are Fluidanimate and Swaptions and three from SPLASH, Radiosity, Raytrace and Water. Fluidanimate and Radiosity have very

TABLE III
PERFORMANCE RESULTS OF THE FLUIDANIMATE BENCHMARK BEFORE AND AFTER MODIFICATION OF CODE

Fluidanimate	**Orig**	**Modified**	**Det**	**Det with Opt5**	**Det with Opt5&6.2**	**Det with Opt5&6.1&6.2**
Exec Time & Overheads (4T)	1239	1148	2382 (**92%**)	1946 (**57%/70%**)	1673 (**35%/46%**)	NA
Exec Time & Overheads (8T)	1327	1172	4612 (**247%**)	3851 (**190%/228%**)	3586 (**170%/205%**)	2646 (**99%/125%**)

TABLE IV
PERFORMANCE RESULTS OF OUR SCHEME AS COMPARED TO KENDO

Benchmark	**Fluidanimate**	**Swaptions**	**Radiosity**	**Raytrace**	**Water**
Results for 4 threads					
Locks/sec	6655543	216979	2666366	120984	377701
Lock frequency	High	Low	High	Low	Low
Chunk size used	2000	1000	2500	800	7000
Original	**26.88**	**27.03**	NC	**63.38**	**35.96**
Non-det	27.22 (1%)	26.68 (0%)	**41.5**	63.53 (0%)	37.2 (3%)
Unopt SW1	46.56 (73%)	29.90 (11%)	58.14 (41%)	68.80 (9%)	52.56 (46%)
DetLock SW1	39.79 (48%)	30.02 (11%)	51.07 (24%)	63.37 (0%)	44.76 (24%)
Unopt SW2	40.25 (50%)	36.29 (34%)	63.68 (55%)	69.88 (10%)	49.39 (37%)
DetLock SW2	38.05 (42%)	**28.92 (7%)**	**49.0 (18%)**	66.72 (5%)	47.15 (31%)
DetLock SW2 with Opt6	**34.59 (29%)**	-	-	-	-
Kendo	41.1 (53%)	34.55 (28%)	81.1 (95%)	**65.11 (3%)**	**43.1 (19%)**
DetLock HW1	28.01 (4%)	**27.96 (3%)**	49.34 (20%)	64.78 (2%)	44.27 (23%)
DetLock HW2	34.73 (29%)	29.48 (9%)	**48.1 (16%)**	**64.29 (1%)**	44.10 (23%)
DetLock HW2 with Opt6	**32.22 (20%)**	-	-	-	-
KendoHW	32.87 (22%)	29.99 (11%)	49.6 (20%)	64.43 (2%)	**43.8 (21%)**
Results for 8 threads					
Locks/sec	22746816	227835	5273035	120965	542817
Lock frequency	High	Low	High	Low	Low
Chunk size used	2000	1000	2500	800	7000
Original	**16.02**	**13.73**	NC	**63.39**	NC
Non-det	16.18 (1%)	13.59 (0%)	**22.9**	63.68 (0%)	**42.6**
Unopt SW1	62.82 (292%)	18.03 (31%)	53.94 (136%)	75.46 (19%)	64.35 (51%)
DetLock SW1	49.36 (208%)	**16.22 (18%)**	**36.57 (60%)**	73.65 (16%)	59.45 (40%)
Unopt SW2	53.3 (233%)	23.92 (74%)	93.61 (308%)	74.22 (17%)	62.28 (46%)
DetLock SW2	39.29 (145%)	16.99 (24%)	47.2 (106%)	73.47 (16%)	58.4 (37%)
DetLock SW2 with Opt6	**29.35 (83%)**	-	-	-	-
Kendo	59.61 (272%)	34.14 (149%)	71.6 (213%)	**65.08 (3%)**	**57.1 (34%)**
DetLock HW1	38.88 (143%)	**15.84 (15%)**	**29.75 (30%)**	63.74 (0%)	**58.0 (36%)**
DetLock HW2	29.52 (84%)	16.55 (21%)	39.4 (72%)	63.76 (0%)	**58.0 (36%)**
DetLock HW2 with Opt6	**24.0 (50%)**	-	-	-	-
KendoHW	28.13 (76%)	16.99 (24%)	37.2 (62%)	**63.61 (0%)**	58.4 (37%)

high number of shared memory accesses, while Swaptions, Water and Raytrace have low shared memory accesses. The benchmarks were run using the Marssx86 [19] simulator, which is a cycle accurate simulator for multicore x86 systems. All benchmarks were run using 4 and 8 cores, where each core is occupied by a different thread. The simulator has been modified to include our hardware-assisted approach.

Kendo updates clocks by using performance counters. The performance counter for retired stores for example, is programmed to give an interrupt after fixed number of stores. So, for example, we can set the performance counter to update the clock after every 2000 stores. The authors of Kendo call this number, the chunk size. For benchmarks, which were used in the Kendo paper, we use the same chunk size, but for other benchmarks, which are Fluidanmiate and Swaptions, we use a chunk size of 2000 and 1000 respectively. Our implementation does not involve performance counters and interrupts, as that would require further change for the simulator and also modification of the kernel. However, we mimic Kendo as close as possible by updating the retired store count after number of stores, which are equal to the chunk size. So actually, the results would be a little bit better than actual as their would be no interrupts. It should be noted that those interrupts can have a significant impact on performance, as the author of Kendo showed that by reducing the chunk size from 2500 to 1000, the interrupt overhead for Radiosity increased from about 10% to 20%. Note that for the Kendo paper, the author only used a chunk size of less than 2000 for only 2 benchmarks out of 10, that is why, we think keeping the chunk size for Fluidanimate and Swaptions, which were not used in the Kendo paper, to 2000 and 1000, is fair enough.

For each benchmarks, we show frequency of locks acquired, followed by time taken by the original program. This is followed by the time taken by the code which includes clock update code for deterministic execution, but does not perform deterministic execution. We call this the Non-det version. This is followed by the performance of DetLock. Here, the SW1 version updates the clock as discussed in the previous sections, that is, it counts every instruction for clock update. However, the SW2 version only counts loads and stores. In most cases, the SW2 version gives better performance. This is because, we work on the intermediate code, which is not really the exact representation of the final compiled code. By counting each instruction, we may increase the error, as instructions are much more in number than only loads and stores. Next in the table, we show the performance of Kendo. This is followed by the hardware assisted versions of DetLock, with version HW1 counting every instruction for clock update while HW2 counting only loads and stores. Finally, we show the results of hardware assisted Kendo (KendoHW).

The original time for Radiosity and Water (with 8 threads) was not calculated (indicated as NC in the table) due to a bug causing it crash during simulation. In this case, the percentage overheads in the table are calculated with respect to the non-deterministic version with clock updating code (Non-det).

For benchmarks with high amount of shared memory accesses, DetLock outperforms Kendo. The reason that DetLock performs better for such benchmarks is that the waiting time is less, as DetLock updates the clock much more frequently than Kendo, whereas Kendo only updates the clock after given number of chunks. For example, for Fluidanimate, the improvement is 189% for 8 threads (From 272% to 83%). As far as benchmarks with low frequency of shared memory accesses are concerned, DetLock performs better with Swaptions. This is because in Swaptions, such compute intensive loops are found for whom clock count can be updated ahead of time, that is, before executing those loops, thus increasing the deterministic execution performance by reducing waiting time for threads trying to acquire locks. On the other hand, there is not much opportunity to significantly update the clock ahead of time for Water and Raytrace. Therefore, for Water, DetLock performs at par with Kendo, while for Raytrace, Kendo performs slightly better than DetLock. We suspect that the reason Kendo performs slightly better for Raytrace is because the clock used by Kendo is more accurate, as Kendo counts the actual stores happening during run time, while we count loads and stores or instructions at compile time and that too on the intermediate code.

As far as the hardware assisted versions of Kendo and DetLock are concerned, the DetLock version is at par with the Kendo for all benchmarks except Fluidanimate and Swaptions. For Fluidanimate, DetLock improves performance by 26% for 8 threads. Even for a Swaptions, which has low lock frequcency, DetLock performs a little better as compared to Kendo, due to its ability to update clocks ahead of time. Unlike the original Kendo version, which updates clock after a certain number of stores, which depends on the selected chunk size, our hardware assisted Kendo version updates the clock after each store. This is why it performs so much better than the original Kendo version.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we described the design and implementation of a user-level leader/follower based fault tolerance scheme for multithreaded applications running on multicore processors. We use two different techniques to ensure deterministic shared memory accesses by the replica. The first one is record/replay technique where one replica logs the order of shared memory accesses and the second one is deterministic multithreading, where the redundant processes do not need to communicate with each other for ensuring deterministic shared memory accesses. From experimental evaluation, we see that record/replay is faster and more scalable than deterministic multithreading at the cost of more memory on benchmarks with large shared memory accesses. On benchmarks with small to moderate shared memory usage, the performance of these two is comparable.

For 4 threads, record/replay has an overhead of less than 25% for all benchmarks, while with deterministic multithreading, the overhead is also less than 25% for benchmarks with little to moderate shared memory accesses, but can reach 56% for those with high shared memory accesses. The overhead of deterministic multithreading was significantly reduced by applying various optimizations. For example, for Fluidanimate, the overhead was brought down from 77% to just 33% for 4 threads using these optimizations.

To improve performance and scalability, we implemented a hardware extension to aid in deterministic multithreading. As compared to Kendo, our deterministic multithreading Scheme (DetLock) showed overhead reduction up to 189% for 8 threads. Even when Kendo is assisted with hardware, the hardware assisted DetLock version can improve performance up to 26% over Kendo.

ACKNOWLEDGMENT

This research is supported by Artemis through the SMECY project (grant 100230).

REFERENCES

[1] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.
[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.
[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. PACT '08, pages 72–81.
[4] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. HPCA '11, pages 333 –334, feb.
[5] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.
[6] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, ASPLOS '10, pages 77–90.
[7] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. ASPLOS '09, pages 97–108.
[8] H. Mushtaq, Z. Al-Ars, and K. Bertels. DetLock: Portable and Efficient Deterministic Execution for Shared Memory Multicore Systems. MuCoCos '2012.
[9] H. Mushtaq, Z. Al-Ars, and K. Bertels. Efficient and Highly Portable Deterministic Multithreading (DetLock). Computing, vol. 96, pp. 1131-1147, 2014.
[10] H. Mushtaq, Z. Al-Ars, and K. Bertels. Efficient Software Based Fault Tolerance Approach on Multicore Platforms. DATE '2013.
[11] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. SOSP '11, pages 327–336.
[12] V. Weaver and J. Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? FHPM '10.
[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. ISCA '95, pages 24–36.

[14] R. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, 1(1):17 –22, March 2001.

[15] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + dmr: practical and low-overhead permanent fault detection. ISCA '11, pages 201–212.

[16] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[17] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. SIGMETRICS '10, pages 155–166.

[18] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. ICS '11, pages 359–368.

[19] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A full system simulator for multicore x86 CPUs. DAC '11, pages 1050–1055.

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. PLDI '07, pages 89–100.

6

WCET CALCULATION USING DETERMINISTIC MULTITHREADING

SUMMARY

In this chapter, we use a model checking based approach to calculate the WCET, where we apply optimizations to reduce the number of states stored by the model checker. Furthermore, we use deterministic shared memory accesses to further reduce calculation time, memory and number of states needed for calculating WCET. By optimizing the model checking code, we are able to complete benchmarks which otherwise have state explosion problems. Furthermore, by using deterministic execution, we significantly reduce the calculation time, memory and states needed for calculating WCET with a negligible increase in the calculated WCET for a multicore system. Lastly, unlike other state-of-the-art approaches, that perform binary search to search the WCET by running several iterations, our method calculates WCET in just one iteration.

By optimizing the model checking code, we are able to complete benchmarks which otherwise have state explosion problems. Furthermore, by using deterministic execution, we significantly reduce the calculation time (up to 158x), memory (up to 89x) and states needed (up to 188x) for calculating WCET, with a negligible increase (up to 4%) in the calculated WCET for a multicore system with 4 cores. Lastly, unlike other state-of-the-art approaches, that perform binary search to search the WCET by running several iterations, our method calculates WCET in just one iteration. Taking all these optimizations into consideration, the gain in speed of WCET calculation is from 1775x to 2471x for 4 threads.

This chapter is based on the following papers.

1. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Accurate and efficient identification of worst-case execution time for multicore processors: A survey*, Design and Test Symposium (IDT), 2013 8th International, 16-18 Dec. 2013

2. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Calculation of Worst-Case Execution Time for Multicore Processors using Deterministic Execution,* Submitted to PATMOS 2015.

Accurate and Efficient Identification of Worst-Case Execution Time for Multicore Processors: A Survey

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

***Abstract*—Parallel systems were for a long time confined to high-performance computing. However, with the increasing popularity of multicore processors, parallelization has also become important for other computing domains, such as desktops and embedded systems. Mission-critical embedded software, like that used in avionics and automotive industry, also needs to guarantee real time behavior. For that purpose, tools are needed to calculate the worst-case execution time (WCET) of tasks running on a processor, so that the real time system can make sure that real time guarantees are met. However, due to the shared resources present in a multicore system, this task is made much more difficult as compared to finding WCET for a single core processor. In this paper, we will discuss how recent research has tried to solve this problem and what the open research problems are.**

I. Introduction

For a long time, single core processors ruled the desktop and embedded market. The popularity of the single core processors could be attributed to the portability they provided. A program written for one processor, could be ported to the faster version of the same processor without changing a single line of code. However, at one point, it was no more possible to build faster single processors due to the huge amount of power they would need. That is the point, where multicore processors came into existence, as they are more power efficient. Nowadays, multicore processors are common in desktops, laptops and mobile phones. However, industries which use mission critical embedded software, such as avionics and the automotive industry have been reluctant to employ multicore systems. The reason being that such software also needs to meet timing deadlines for real time performance. For guaranteeing real time performance, the real time scheduler needs to know the worse-case execution time (WCET) of each task. Finding a good estimate (less pessimistic) of WCET, of a task is much simpler if it runs on a single core processor than if it runs on a multicore processor concurrently with other tasks. This is because those tasks can share resources, such as shared cache or shared bus, and/or may need to concurrently read and/or write shared data.

Recently, there has been an increasing interest to solve the problem of finding WCET for tasks running on multicore processors, from hardware solutions to software solutions for Commodity Off The Shelf (COTS) processors. In this paper, we discuss the research done in this context and also point out the open issues. In Section II, we provide the necessary background to help reader understand the problem of WCET. This is followed by Section III which discusses the WCET

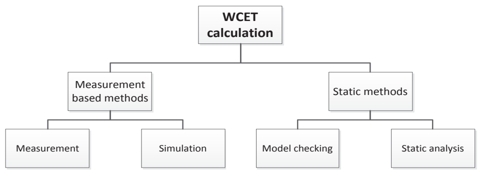

Figure 1. Methods used for WCET calculation

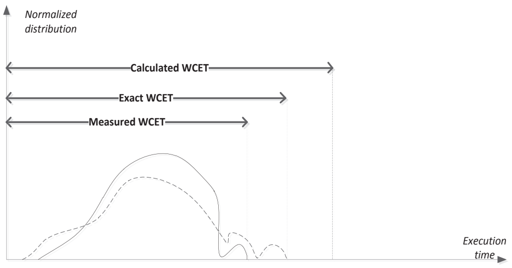

Figure 2. Measurement based vs static methods

calculation techniques employed for single core processors. Next, we discuss the research that has been done for calculating WCET of multicore processors in Section IV. This is followed by Section V on open issues. We finally conclude the paper with Section VI.

II. Background

Multicore processors can be useful in embedded systems, such as automotive systems, as that would mean that software could be made more centralized. This translates to less cable usage in cars, and therefore less fuel consumption, as more cable length is directly proportional to fuel consumption in cars. Moreover, with processors with more cores, more functionality could be added, for example, we could have an improved braking system, which uses more sensors [27].

Mission critical embedded systems perform hard real time tasks, which need to complete within a certain time period. To be able to guarantee that those tasks finish within that time period, their WCET should be known. For single core processors, techniques to find WCET are well known and there are several tools available to perform that. Those techniques and tools can be found in the literature [30].

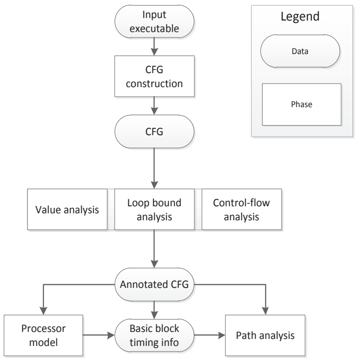

Figure 3. Steps for WCET calculation using static analysis

As seen from Figure 1, there are two main methods of finding WCET, measurement based methods and static methods. In measurement based methods, the execution time is measured either through direct measurement or simulation of the code by giving different inputs. The obvious drawback of this method is that the WCET can be underestimated in this way, as not all possible paths could be tested with the limited number of inputs. This fact is shown in figure 2, where the curve for the measurement-based experiments is shown with a solid line, while the curve with all possible inputs possible is shown with a dotted line. To overcome this, one could put a safety margin over the measured WCET. However, the safety margin is still just a guess and the picked WCET could end up less than the real WCET. One way to have better estimations is to measure the worst-case execution time of each basic block and then try to find the path with the worst-case time by adding the time taken by these blocks. However, this would only work for very simple processors. In the presence of advanced features, such as pipeline, branch predication, out-of-order execution and caches, this would not work. In the presence of these advanced features, the worst-case execution time of a block is dependent on the path followed by the program. For example, the cache misses in a basic block will be dependant upon from which path the program reached that basic block.

Due to the fact that measurement based methods underestimate WCET, we limit ourselves to only static methods. There are two major kinds of static methods used for calculating WCET, namely static analysis and model checking. The combination of these two is also employed in some cases. The details of these methods is discussed in detail in the next two sections.

The major steps taken in calculating WCET are shown in Figure 3 [1]. The input executable is read to construct a control flow graph (CFG). Afterwards, control-flow analysis (CFA) is performed which performs steps such as removing infeasible paths, trying to find loop bounds and determine frequencies of execution of paths, etc. At that point, the user could also provide information such as loop bounds which could not be found by CFA, or known values at certain locations in the program, so that CFA can more precisely find infeasible paths. The annotated CFG with the micro-architectural analysis is then used to find the WCET of each basic block. Finally path analysis is performed to find the WCET. Since the goal is to calculate a WCET which is at least equal to the real WCET, the calculated WCET is almost always greater than the real WCET, as shown in Figure 2. Therefore, the quality of a tool measuring WCET is assessed by how close WCET it calculates with respect to the real WCET, in other words how much tighter the WCET it calculates.

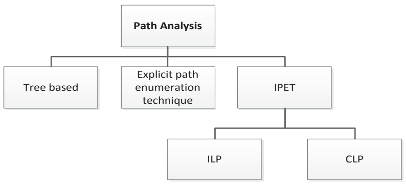

Figure 4. Path analysis methods used for WCET calculation

III. WCET FOR SINGLE CORE PROCESSORS

While the main focus of this paper is on finding WCET for multicore processors, it is first important to discuss the techniques applied to single core processors. This is because, even for single core processors, finding WCET is not straight forward, as typical single core processors are designed to have good average-case execution time, through features such as pipelining, cache memories, out-of-order execution, speculation and branch prediction. All these features, make accurate timing analysis a difficult problem [4].

These performance-enhancing features, also introduce timing anomalies. For example, one may assume that it would be safe to use a cache miss time for WCET calculation. However [21] showed that for out-of-order execution, it is possible that in some cases a cache hit would increase the time as compared to a cache miss.

There exist two main techniques for finding WCET for single core processors, namely static analysis and model checking. Static analysis is more computationally efficient in finding the WCET as compared to model checking, as model checking can have state-space explosion problems. However, model checking can find tighter WCET estimates. This is because, model checking can more accurately model a processor, while static analysis can just approximate the processor model, as it needs to find the WCET without actually running the program. These two techniques can be combined though to achieve the best results. In Section III-A, we will discuss related work using static analysis, while in Section III-B, we will discuss techniques which employ model checking for finding WCET.

A. Static analysis

Static analysis techniques try to find the WCET without actually running the program. Since they do not run the program, they need to approximate the processor model. Therefore, static analysis techniques are divided into two steps. First step is performing CFA on the CFG, and performing value analysis to find loop bounds, values to eliminate infeasible paths and addresses to help in finding cache hits and misses, followed by processor modeling to obtain the WCET of each basic block in the program. There are three techniques to do this, abstract interpretation (AI), integer linear programming (ILP) and constraint logic programming (CLP). The second step is to find the WCET using WCET of the basic blocks. Different techniques employed for this purpose (Path analysis) are shown in Figure 4.

1) Tree based: The tree based method for path analysis traverses the CFG in bottom-up fashion, combining the WCETs of the basic blocks along the way (see [30] for more detail). This method is quite efficient but suffers from some limitations. For example, it is not possible to represent goto statements. Also, it is difficult to eliminate infeasible paths. An example of a paper using this technique is [9], which employs this technique for a processor with pipeline, branch prediction and an instruction cache.

2) Explicit path enumeration technique: This technique tries to find the longest path in terms of execution time by looking for all the possible paths in the program. It first tries to eliminate all infeasible paths in the program. This method suffers from low performance, as the number of paths that need to be examined increases exponentially with the program size.

3) ILP: Due to the problem associated with explicit-path-enumeration technique, authors in [19] propose integer linear programming (ILP) to solve the WCET problem implicitly. That is why this techniques is known as implicit-path-enumeration technique (IPET). Equation 1 is the basic equation of calculating WCET with this technique. Here c is the cost of basic block i and x is the number of times that basic block is executed. The WCET is given by finding the maximum value of Equation 1.

$$\sum_{i=1}^{n} c_i x_i. \tag{1}$$

The authors of [19] extended their work to also account for architectures with instruction caches in [20]. Both modeling of the instruction cache (processor modeling) and calculating of WCET (path analysis) is done using ILP. A basic block is further divided into line blocks, where each line block represents contiguous instructions which use the same cache line. Also, information is kept for a line block whether it incurs a cache miss or is a cache hit. This method might work for simple models, but for more complex processor architectures, which include pipelining, speculative execution, branch prediction and out-of-order execution for example, it becomes prohibitively difficult to use ILP due to its restrictive nature. For those purposes, Abstract Interpretation (AI), which is discussed next, is much more feasible.

4) AI+ILP: Abstract Interpretation (AI) is a dataflow technique to approximate model of a processor. AI can be used for example to get a set of possible values for a variable. However, since AI is an approximate method, it might also include values in a set, which would not occur in the program. Therefore, techniques using AI overestimate WCET at the cost of finding WCET in less time as compared to model checking.

That is why authors in [29] separate processor modeling and path analysis steps. For processor modeling, they use AI. Through AI, they model pipeline and caches. Through AI, they can classify an instruction as always hit, always miss, persistent (miss for first time and then always hit) or unclassified. In the case of unclassified, both scenarios are considered, that is cache hit and cache miss, as previously discussed that due to timing anomalies, it is not enough to consider cache miss as the worst case scenario.

While [29] checks for always miss, always hit and unclassified instructions in a global scope, [15] also consider local scopes like loops and functions. They argue that the same cache line block used in different scopes might not interfere with each other and therefore would be mutually exclusive, so in this way we could have blocks which are classified as persistent only in that local scope. This method reduced estimates of the WCET by upto 74%.

5) CLP: Another alternative of processor modeling and path analysis using ILP is to use constraint logic programming (CLP). The drawback of ILP is that we are limited to only using linear constraint with ILP, and for representing disjunction, we have to duplicate a block. For example, if a block can be reached from two different paths, it has to be duplicated into two blocks, each having a different WCET, but with CLP, we can actually define through constraint equations the value of WCET values for that block for different paths. Authors in [22] showed that using CLP significantly reduced WCET calculation time as compared to ILP, as there are much less blocks required due to the ability of representing disjunctions through constraint equations.

B. Model checking

The problem with static analysis is that it is an approximate method, due to the approximate nature of the processor modeling steps. With model checking on the other hand, we can build a more concrete model of a processor, and therefore have tighter WCET estimates. For example, when cache accesses cannot be classified with AI, we have to check execution time with both cache miss and cache hit. On the other hand, with model checking, some of those instructions which were unclassified in AI, could be classified, thus tightening up the WCET estimates. [14] is an example which supports model checking for finding WCET for Java processors. The authors use UPPAAL [3] model checker for that purpose. The authors noticed that model checking was enough for typical tasks in embedded systems. However, for larger applications, it was too slow. The authors recommended that model checking could be combined with static analysis in such a way, that the more important code fragments could be analyzed with model checking while the remainder of the application with static analysis.

[23] also uses a model checker. Instead of using each instruction in the model checker, the authors use basic blocks, thus reducing the number of states for model checking.

[17] combine model checking with static analysis to find WCET. The model checking is useful in deriving loop bounds, which change dynamically. For example, for loop bounds that depend upon two variables, the user just has to feed a range of values of these variables and the model checker can extract the loop bounds from them.

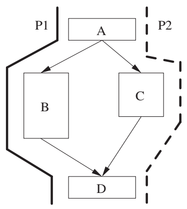

Figure 5. Example of timing anomaly in a multicore processor [32]

IV. WCET FOR MULTICORE PROCESSORS

We discussed previously that single core processors can have timing anomalies in the presence of complex performance enhancing features. Multicore processors have another source of timing anomalies due to shared resources, such as shared cache memory. An example is shown in Figure 5. Let us assume the path ABD is the worst-case path if seen separately. In the presence of shared L2 cache however, ACD might become worst-case path if a thread running on another core evicts more instructions from C than B in the L2 cache. Therefore, whenever analyzing WCET for a multicore, we always need to consider all the tasks running on different cores together, which can significantly increase the complexity of timing analysis.

In Section IV-A, we discuss the WCET calculation methods used for mutlicore systems, which are static analysis and model checking, whereas, in Section IV-B, we discuss techniques that assist in WCET estimation.

A. WCET calculation methods

Like in case of single core processors, WCET calculation techniques can also be divided into static analysis and model checking. These two techniques for multicore systems and their comparison is discussed next.

1) Static analysis: The first work done in this regard was by [32], which extends [29] to a multicore processor with private L1 caches but shared L2 cache. Through AI, this method tries to find out which instructions are always cache hits or always cache hits after the first time. It considers all other instructions as cache misses. This method first checks for L1 cache misses separately. An L1 cache miss implies either an L2 cache hit or an L2 cache miss. The basic idea is to check if the same cache block will be used by another thread running on another core. If that is the case, the basic block is marked as to have an L2 cache miss if the other thread is using that block with a loop, otherwise it is marked as always-except-one-hits. If the cache block is not used by the other core, then it is marked as always-hit. The WCET is found by solving the linear constraints formed by AI. The authors of this paper extend this method to also include data caches in [33]. The drawback of this method though is that it considers the effect of caches in isolation, that is not including performance enhancing features such as branch prediction and speculative execution which can cause timing anomalies. In case of timing anomalies, it is not enough to assume cache miss as the worst case.

To solve this problem [7] include pipelining, branch prediction and speculative execution in their analysis. Although they only consider instruction L2 caches. With timing anomaly in consideration, the timing analysis becomes more complex, as we cannot just assume a cache miss, if we are not sure about a cache access being a cache miss or a cache hit. This is the reason, the authors classify cache accesses as always-hit, always-miss and unclassified. For unclassified, both a cache hit and a cache miss are tried to find out the WCET.

[13] uses a technique similar to [7] but improves WCET calculation by employing bypassing of caches. Bypass of a load instruction for example, for a cache level means that if there is a miss, the memory block would not be brought into the cache, while if there is a cache hit, age of no cache block would be altered. In this way, instructions which are rarely used in a program could be bypassed, thus reducing inter-core conflicts and therefore improving timing analysis. The instructions to be bypassed are chosen by the compiler at compile time. [13] only considers bypassing for instruction caches, while [18] does it for both instruction and data caches. It has to be noted though that not every processor has a bypass instruction and therefore this method is not portable.

2) Model checking: Besides, static analysis, model checking is also a viable approach for calculating WCET for a multicore processor. We can either use model checking alone or combine it with static analysis. When model checking is used alone, both the processor modeling and path analysis is done using the model checker. The user can query the model checker with a guess WCET, to see if the maximum time calculated by the model checker is less than or equal to the guessed WCET. The guess is refined until the WCET is matched with the maximum time calculated by the model checker.

[31] uses model checking to estimate WCET. The model checking language used is PROMELA, which is the language of SPIN [2] model checker. The approach works for shared L2 caches. The authors show that using model checking improves the tightness of WCET as compared to static analysis only approaches. This is because model checking can check every possible interleaving of threads running on different cores, and therefore some cache accesses which cannot be classified as cache miss or cache hit, can be properly classified with a model checker. One problem with using a model checker is the state-space explosion problem. The authors of [31] tried to reduce this by first finding L1 and L2 cache hits and misses for a task by assuming it is running on a single core processor. This information is then imported for model checking the real scenario, that is tasks running on a multicore processor. In this way, only the L2 hits need to be taken care of, as L2 misses would still be misses on a multicore.

[8] combines static analysis with model checking to calculate WCET. AI is used to model the shared cache, but model checking is used to model the shared bus, which is the bus that is used to read from and write to the main memory. The main reason of using model checking for the shared bus is because it is much simpler to model it with a model checker as compared to modeling it with AI. Furthermore, since it is more accurate, it also gives tighter WCET estimates.

[12] uses UPPAAL [3] to model a multicore system with private L1 caches and a shared L2 cache. This method also works for tasks communicating with each other through shared memory using spin locks. Since, each instruction is modeled,

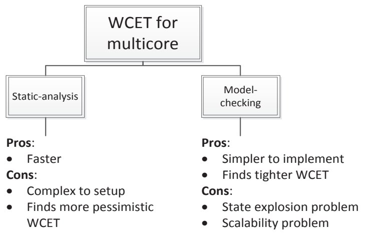

Figure 6. Model checking vs static analysis comparison

the state space is large, and therefore this method only works for small programs. Even for those programs, a WCET can only be calculated with two cores. For more cores, state space explosion is observed.

[6] uses model checking and static analysis. This method supports shared memory communication among the tasks. A program is divided into communication and execution slices. At the start of an execution slice, data is loaded into the private caches of the cores and at the end of the communication slice, data is put back in the main memory. State-space explosion occurs when there is too much communication involved. Also, this method slows down execution, as data has to be read from and written to the main memory at each execution and communication slice.

3) Model checking vs static analysis: Figure 6 compares static analysis with model checking for finding WCET for multicore processors. We can see that static analysis is faster but finds more pessimistic WCET as compared to model checking based approaches. Moreover, it is also more difficult to implement. The problem with model checking is that it suffers from scalability problem, as with more cores, there are more states possible, thus causing state-space explosion for larger programs. The good thing though is that model checking can be aided by static analysis to reduce those states, as done by some papers discussed in Section IV-A2. However, none of those papers used a processor with more than 2 cores, suggesting that even by combining static analysis with model checking, it is still difficult to find a scalable method.

B. Assisting WCET estimation

Hardware approaches can ease in estimating tighter WCET. For example, [10] proposes hardware mechanism to allow execution of synchronization operations such as mutex locks in bounded time. The logic for the hardware synchronization primitives (such as test-and-set, fetch-and-increment/decrement) is nested in the memory controller.

Cache locking and cache partitioning [28] can make the task of WCET calculation much easier. Cache partitioning means that the tasks running on different cores use a separate portion of the shared cache, while cache locking allows a user to load certain data in the cache and lock it, that is, prevent it from being replaced. The benefit of cache partitioning is that one could perform WCET calculation for tasks running on separate cores separately. While cache partioning can ease the calculation of WCET, it can also reduce performance, as due to less cache space available to a task, more cache misses could occur.

[16] proposes synchronized cache management to ease finding a tighter WCET. This is done by using page coloring. Physical pages of different colors do not cause cache conflict. Moreover, there are limited number of pages of the same color. Accesses within the same colored memory by different cores cause conflict only when the number of cache ways are exhausted. The authors view each color as a shared resource, where a lock is required to access that shared resource. For locking, the authors implemented a synchronization protocol.

[26] propose an interference-aware arbiter, through which the maximum time to access a shared resource by a hard real-time task (HRT), such as shared memory has an upper bound. The system assumes that both HRT and non-real time (NRT) tasks are running concurrently on the system and makes sure that the access to a shared resource by an HRT is bounded in time to ease WCET calculation.

V. Open issues

For single core processors, there are several tools available for estimating WCET of tasks as discussed in [30]. However, there is no such tool available yet for multicore processors, as we saw that timing analysis for multicore processors is much more complex due to increased number of states possible due to access of shared resources. Here, we discuss the still open issues for solving the problem of estimating WCET for multicore systems.

A. Scalability and precision

Although the scalability of static analysis is better as compared to model checking, the static analysis methods are still not very scalable, as the possible number of states with a multicore processor is still much more than that of a single core one. There are no papers yet that use more than two cores for experiments. All of the static analysis approaches that we discussed use ILP for path analysis. It would be interesting to use CLP instead, because [22] showed that CLP is much faster than ILP on single core.

A combination of model checking and static analysis methods could represent the most appropriate solution. One way to solve the scalability issue would be to only perform model checking on the compute-intensive part of the code and use static analysis for the rest.

B. Synchronization

Almost all the methods that we discussed for finding WCET on multicore processors ignore the problem of data sharing between the cores, and those that do consider it have some limitations. [6] writes back data to the main memory after every communication slice and reads it back from the main memory at the start of each execution slice, thus incurring a much larger overhead as compared to keeping the shared data in cache. [10] describes a hardware approach to bound the time of synchronization operations, but the obvious drawback of this method is that it needs hardware modifications, thus impacting portability.

One possible solution is to use determinsitic execution [25] [24], where locks for shared memory access are acquired in such a way that there is only one schedule possible. Although this method ensures determinism of shared memory accesses for only a given input, [5] showed that even when an exhaustive set of inputs is considered, deterministic execution can have a smaller schedule space than non-deterministic approaches. However, to employ this method, the problem of global clock reading that [25] and [24] employ would need to be solved first, as global clock reading can cause cache evictions and therefore increase cache coherence activity.

VI. Conclusion

In this paper, we discussed the challenges of finding WCET for multicore processors and discussed some recent approaches in that direction. However, none of the existing approaches have been tried and tested for more than two cores, thus raising the concern of scalability of such approaches. Also, most of these approaches ignore the fact that data could be shared between the cores. Those that do, suffer from performance or portability problems. We also gave suggestions on how this scalability and synchronization problem could be solved.

Acknowledgment

This work is carried out under the BENEFIC project (CA505), a project labeled within the framework of CATRENE, the EUREKA cluster for Application and Technology Research in Europe on NanoElectronics.

References

[1] http://compilation.gforge.inria.fr/2010_12_Aussois/programpage/pdfs/MAIZA.Claire.aussois2010.pdf

[2] http://spinroot.com/spin/whatispin.html

[3] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[4] Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In *Perspectives Workshop: Design of Systems with Predictable Behaviour, 16.-19. November 2003, volume 03471 of Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl*, 2004.

[5] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails? In *2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[6] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In *Proceedings of the 25th International conference on Architecture of Computing Systems*, pages 98–110, Berlin, Heidelberg, 2012.

[7] S. Chattopadhyay, C.L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *Proceedings of the 18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 99–108, 2012.

[8] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software &#38; Compilers for Embedded Systems*, pages 6:1–6:10, New York, NY, USA, 2010.

[9] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proceedings of the of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, 2001.

[10] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. Time analysable synchronisation techniques for parallelised hard real-time applications. In *Design, Automation Test in Europe Conference Exhibition 2012*, pages 671–676.

[11] Sylvain Girbal, Miquel Moretó, Arnaud Grasset, Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, and Sami Yehia. On the convergence of mainstream and mission-critical markets. In *Proceedings of the 50th Annual Design Automation Conference*, pages 185:1–185:10, New York, NY, USA, 2013.

[12] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards wcet analysis of multicore architectures using uppaal. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 103–113. Österreichische Computer Gesellschaft, 2010.

[13] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 68–77, 2009.

[14] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based wcet analysis. In *In Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[15] Bach Khoa Huynh, Lei Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.

[16] Christopher J. Kenna, Jonathan L. Herman, Bryan C. Ward, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*, 2013.

[17] Sungjun Kim, Hiren D. Patel, and Stephen A. Edwards. Using a model checker to determine worst-case execution time. Technical report, Columbia University Computer Science Technical Reports, 2009.

[18] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, page 2283, Toulouse, France, 2010.

[19] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 456–461, New York, NY, USA, 1995.

[20] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. In *ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS*, pages 257–279, 1999.

[21] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[22] Amine Marref and Guillem Bernat. Predicated worst-case execution-time analysis. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 134–148, Berlin, Germany, 2009.

[23] Alexander Metzner. Why model checking can improve wcet analysis. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer Berlin Heidelberg, 2004.

[24] H. Mushtaq, Z. Al-Ars, and K. Bertels. Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 721–730, 2012.

[25] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, March 2009.

[26] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[27] H. Shah, A. Raabel, and A. Knoll. Challenges of wcet analysis in cots multi-core due to different levels of abstraction. In *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, 2013.

[28] V. Suhendra and T. Mitra. Exploring locking partitioning for predictable shared caches on multi-cores. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 300–303, 2008.

[29] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

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

[31] Lan Wu and Wei Zhang. A model checking based approach to bounding worst-case execution time for multicore processors. *ACM Trans. Embed. Comput. Syst.*, 11(S2):56:1–56:19, August 2012.

[32] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 80–89, 2008.

[33] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 455–463, 2009.

Calculation of Worst-Case Execution Time for Multicore Processors using Deterministic Execution

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

Abstract—**Safety critical real time systems need to meet strict timing deadlines. We use a model checking based approach to calculate the WCET, where we apply optimizations to reduce the number of states stored by the model checker. Furthermore, we used deterministic shared memory accesses to further reduce calculation time, memory and number of states needed for calculating WCET. By optimizing the model checking code, we were able to complete benchmarks which otherwise were having state explosion problems. Furthermore, by using deterministic execution, we significantly reduced the calculation time (up to 158x), memory (up to 89x) and states needed (up to 188x) for calculating WCET with a negligible increase (up to 4%) in the calculated WCET for a multicore system with 4 cores. Lastly, unlike other state-of-the-art approaches, that perform binary search to search the WCET by running several iterations, our method calculates WCET in just one iteration. Taking all these optimizations into consideration, the gain in speed was from 1775x to 2471x for 4 threads.**

I. Introduction

Adapting multicore systems to real time embedded systems is a challenging task, as a real time process, besides being error free, must also meet timing deadlines. The real time scheduler needs to know the worst-case execution time (WCET) of each task. Finding a good WCET estimate (less pessimistic) of a task is much simpler if it runs on a single core processor than if it runs on a multicore processor concurrently with other tasks. This is because those tasks can share resources, such as shared cache or shared bus, and/or may need to concurrently read and/or write shared data.

Recently, there has been an increase in interest to solve the problem of finding WCET for tasks running on multicore processors, from on-chip hardware support to software solutions for commodity off the shelf (COTS) processors. But most of those do not take into account the shared memory accesses. In [6], the authors do take into account shared memory accesses, but the state explosion problem of the model checking based approach they use limits the effectiveness of that approach.

In this paper, we investigate, whether deterministic shared memory accesses [7] [8] would reduce the possible number of states used by the model checker and therefore reduce the WCET calculation time. The contributions of this paper are as follows.

- Limiting the state space explosion problem by utilizing deterministic execution when calculating the WCET of a multithreaded program running on multicores using model checking.
- Implementing optimizations to further reduce the size of the state space as well as to get a tighter WCET estimation.
- Using only one iteration to calculate the WCET rather than performing binary search as used by the current state-of-the-art approaches (which requires several iterations).

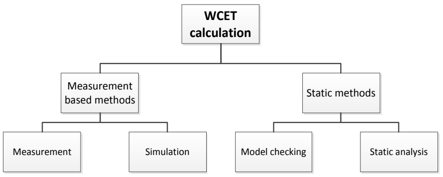

Figure 1. Methods used for WCET calculation

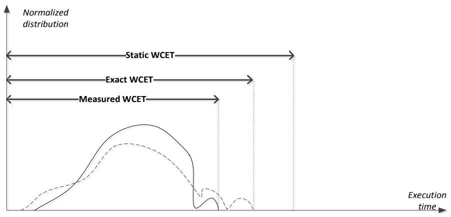

Figure 2. Measurement based vs static methods

In Section II, we discuss the background, while in Section III, we discuss the implementation. This is followed by the Section IV, which discusses the performance evaluation. We finally conclude the paper with Section V.

II. Background

Safety critical real time embedded systems need not only be functionally correct but also meet strict timing deadlines. For this purpose, it is necessary to calculate the WCET of these tasks. However, calculation of WCET is not straight forward for modern processors due to features such as multi-level caches and out of order execution.

There are two methods of calculating WCET, measurement based and static methods as shown in Figure 1. In measurement based methods, we test the runtime by giving different inputs. However, it is often very difficult to test a program with all different inputs. Not checking the program with every possible input might give an underestimated WCET, as shown in Figure 2. A more appropriate approach is to use static methods, which can be classified into static-analysis and model-checking based. In static analysis, rather than running the program with different inputs, all possible paths are statically checked for calculating WCET. In static analysis, often abstract interpretation [13] is used to model the architectural features of a processor using an approximated model. On the other hand, with model checking, one can write code for a precise model of the processor. The result is a tighter WCET, but using more computational overhead.

In this section, we first describe the problem of WCET calculation (Section II-A), and then the description of deterministic execution that can be used to reduce the number of states during model checking (Section II-B).

A. WCET calculation

Modern processors have features such as cache hierarchies and out of order execution, which are meant to improve the average-case execution time of programs running on them. However, these features make it much more difficult to determine a tight WCET. In addition, more complex architectures mean more states for a model checker to keep track of, making it more prone to state explosion problems. Despite these problems, there exist sophisticated tools, such as Chronos [14], that can guess a good WCET for programs running on single core processors. Multicore systems on the other hand have an additional complexity, due to shared resources, such as shared memories. With shared memory, tasks running on different cores also need to synchronize to access the shared data, for example by using locks. This makes it difficult to deduce tight WCET bounds for such systems. Synchronization of shared memory accesses also means many different possible interleaving of the threads are possible, which further aggravates the problem of calculating the WCET. They can have timing anomalies due to shared resources and shared memory accesses. For example, assume that a path ABD is the worst-case path if seen separately, where A, B and D are basic blocks. In the presence of shared L2 cache however, another path, say ACD might become the worst-case path if a thread running on another core evicts more instructions from C than B in the L2 cache. Therefore, whenever analyzing WCET for a multicore, we always need to consider all the tasks running on different cores together, which can significantly increase the complexity of timing analysis.

Recently, there have been several papers published which deal with calculating WCET on multicore processors. A survey of those techniques is given in [12]. Some of those assume that there are no shared memory accesses by the tasks running on the different cores. In other words, they assume that tasks are running embarrassingly parallel to each other. They only cater for the problem of shared L2 cache accesses [10] [11] and the shared bus [5]. Papers like [15] and [16] do consider shared memory synchronization, but they assume simpler processor architectures which do not have any cache, but only scratchpad memories. Such kind of processors are not mainstream and require special programming techniques, since the scratchpad memories have to be manually managed by the programmer.

[6] considers both cache coherence as well as synchronization operations such as spin locks for shared memory accesses. The authors use UPAAL [3] based model checking for that purpose. They do take into account shared memory accesses, but their solution suffers from state explosion problem even for very simple programs. [9] also uses model checking but do not support synchronization operations. [17] recently proposed a mathematical model to determine WCET of multicore systems with caches and cache coherence using abstract interpretation. However, they still do not consider cache coherence that is generated due to accessing the shared synchronizing objects. Moreover, they do not perform any evaluation.

Table I. COMPARISON OF DIFFERENT TOOLS

Tool	Method Used	L2 cache	Shared data	L1 CC
[10][11]	Static analysis	+	-	-
[5]	Static analysis & Model checking	+	-	-
[15][16]	Static analysis	-	+	-
[9]	Model checking	+	-	-
[6], This	Model checking	+	+	+

All the tools described above are shown in Table I, along with the methods they use and the platforms they are made for. There is only one tool [6] which considers shared data on systems with L1 cache coherence (L1 CC). However, as explained before, it is too slow as it suffers from state explosion problems even for very small programs. Our tool improves upon that.

In this paper, we investigate whether model checking overhead used for calculating WCET can be reduced using deterministic shared memory accesses [7] [8]. We use the SPIN model checker [1] with its associated language PROMELA for model checking. Moreover, unlike [6], which uses a very simple example, we use real benchmark programs written in C. We use Chronos [14] to compile those programs into assembly and also to construct the control flow graph (CFG). The assembly code and CFG are then used to generate the PROMELA code for model checking to calculate WCET.

B. Deterministic execution

Multithreaded programs have a frequent source of non-determinism in the form of shared memory accesses. Due to this, multithreaded programs can have many possible thread interleavings, which makes it difficult to find WCET of such programs. We can reduce this interleaving altogether if we know the input of the program and perform deterministic execution. Deterministic execution would make sure that the threads perform shared memory synchronization always in the same sequence. Even for multiple inputs, we can still reduce the possibilities as explained by [4].

One such algorithm for deterministic execution is Kendo [7], which uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least logical clock value gets the lock. For example, Thread 1 will be unable to acquire a lock when its logical clock (1029) is higher than that of Thread 2 (329). But, as soon as Thread 2's clock get past 1029, Thread 1 may acquire the lock. With DetLock [8], it was shown that updating clocks ahead of time improves the performance as compared to Kendo. Therefore, in this paper we also update the clocks ahead of time.

III. IMPLEMENTATION

In this section, we discuss the implementation of our tool. Firstly, in Section III-A, we discuss the architecture of the processor used. Next, in Section III-B, we discuss our method of finding WCET using deterministic execution, while in Section III-C, we discuss an optimization applied to reduce

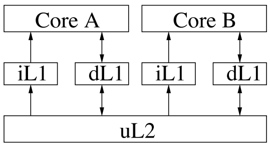

Figure 3. Architecture of the processor used

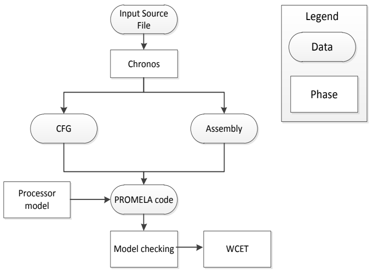

Figure 4. Steps for WCET calculation using static analysis

the calculated WCET with deterministic execution. Finally in section III-D, we describe our method of model checking which avoids performing binary search to calculate WCET, as done by other state of the art approaches.

A. *Processor architecture*

Since, the focus of this paper is to see how much reduction in analysis time we get by using deterministic execution, we assume a simple processor model, which is that every instruction takes one cycle and an L1 cache miss takes 10 cycles, while an L2 cache miss takes 80 cycles. Moreover, a taken branch causes extra 3 cycles. The architecture of the processor is shown in Figure 3. There are separate L1 caches for instruction and data, while the L2 cache is shared. We also assume that there are as many read ports for instruction cache as the number of cores. For cache coherence, we use the MESI cache coherence protocol. We also assume that every shared memory access takes place within a spinlock. For checking whether a certain memory access can cause a cache miss, we check the memory addresses. We check for shared L2 cache misses only for instructions, while assuming all the data is already there in the L2 cache. This assumption is not unreasonable since a typical L2 cache can be large enough to accommodate data for one loop cycle of a real time process. Similarly, for the non-shared data, we assume it has already been brought to the L1 caches of the cores, since the benchmarks we used are small enough to accommodate the local data in the L1 caches of the cores.

B. *WCET calculation*

As shown in Figure 4, we used the Chronos tool to extract the CFG and assembly code. This CFG and assembly code is then used to generate part of the code for our PROMELA code used to calculate the WCET.

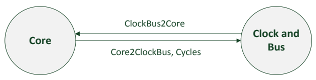

Figure 5. Block diagram of communication between a core process and the *clock and bus* process

We have two kinds of processes in our model checker. There is a core process for each of the cores while there is a *clock and bus* process that represents the processor clock and the shared bus which manages cache coherency. Figure 5 shows the communication channels between the *core and bus* process and a core's process. Through the *ClockBus2Core*, the *clock and bus* process tells a core to either go ahead or wait. A core's process on the other hand sends the number of clock cycles it needs to advance to the *clock and bus* process along with other information on the *Core2ClockBus* channel, such as the address of a shared memory access. Note that the only shared memory access we allow in our model are the shared mutexes and shared variables within locks.

```
atomic {
min_clock = get_minimum_clock();
for( pid = 0; pid < NUM_OF_PROC; i++ ) {
    if( clock[pid] == min_clock )
        advance( pid );
}}
```

Listing 1. Pseudo-code to advance clock cycles

```
 1 atomic {
 2 line = get_cache_line_of_addr(addr);
 3 if (!in_l2_cache[line]) {
 4     in_l2_cache = true;
 5     st[line] = clock[pid] + l2_mt;
 6     wait_for_cycles( pid, l2_mt + l1_mt );
 7 }
 8 else if ( clock[pid] <= st[line] ) {
 9     to_wait = st[line] - clock[pid] + l1_mt;
10     wait_for_cycles( pid, to_wait );
11 }
12 else
13     wait_for_cycles( pid, l1_mt );
14 }
```

Listing 2. Pseudo-code to access L2 cache for instructions

Since a model in which we explicitly synchronize each thread at each clock cycle is quite costly, we have devised a method to significantly reduce the overhead without introducing errors. The C-style pseudo-code for that purpose, which is part of the *Core and Bus* process, is shown in Listing 1. Only the cores with the minimum clocks are allowed to progress, while those which have advanced ahead have to wait. In this way, we make sure that the clocks of the cores are synchronized and yet avoid the overhead of explicit synchronization. In case of accessing instructions from the L2 cache, the C-style pseudo-code to make sure the cores progress properly, is shown in Listing 2. This code is part of a core's process. For example, if a core A experiences L2 cache miss for a cache line, the next core B reading the same cache line would read it from the L2 cache, as the core A would have already brought it into the L2 cache. However, since core B would access that cache line in a later time, to make sure this is properly modeled, we save that value in *st* (line 5), and the clock of core B is advanced using that value if its clock was less than that of core A (line 8). Here *l1_mt* is the L1 cache miss penalty while *l2_mt* is the L2 cache miss penalty. We also use a simplified cache coherence model for shared data which is read-modified only within locks, since only one core would be reading-modifying that data.

We check the WCET, both with and without deterministic execution. For deterministic execution, we used a hardware based comparator, as totally software based deterministic execution would cause substantial cache coherence activity due

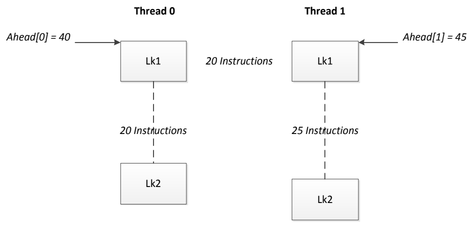

Figure 6. Optimization to improve deterministic execution, WCET and WCET calculation time

to reading the shared clocks for deterministic execution. Each core writes to one of the input registers of the comparator. The comparator writes 1 to the output register whose corresponding input register contains the smallest clock value, while writing 0 to all the other output registers. In this way, a thread can know, whether to acquire a lock by just reading the value of its corresponding output register. In case of two or more input registers having the smallest value, the one with the least index writes 1 to its corresponding output register. Through this hardware, there is no need for threads to read other clocks and also no overhead is incurred due to cache invalidations that occur for maintaining cache coherence.

Since for deterministic execution, we assume a hardware based mechanism, to have a fairer comparison between the deterministic and non-deterministic methods, we also compare the deterministic execution with a method where we assume a hardware based synchronization mechanism, that is, where a lock could be acquired immediately, that is without the overhead of cache coherency for compare and swap operation on a shared variable.

C. Optimization of deterministic execution

We use the DetLock mechanism of updating clock for deterministic execution. One limitation of that method is that a thread cannot update its clock ahead of time if its waiting for a lock. For a program with two threads, Thread 0 can acquire a lock only when its clock is less than or equal to that of Thread 1 as shown by the condition below, where *dt[0]* and *dt[1]* are logical clocks for Thread 0 and Thread 1 respectively.

$dt[0] \leq dt[1]$

To overcome the above mentioned limitation, besides the logical clock, we introduce two more variables for each thread. These two variables are *ahead* and *nl*, where *ahead* is used by a thread waiting for a lock to tell other threads, that its not going to acquire a subsequent lock, which is different from the one it is waiting for, at least for the amount of instructions assigned to *ahead*. On the other hand, *nl* is the number of the lock, or more precisely the address of the mutex in question. The formula for lock acquisition for Thread 0, now changes to the following.

$dt[0] \leq (dt[1] + ahead[1] \times (nl[1] \neq nl[0]))$

This mechanism can be more easily understood by Figure 6. Here if Thread 0 has reached the place where it is trying to acquire Lk2. With the DetLock only approach, it would not be able to acquire that lock, until Thread 1 has unlocked Lk1. However, with this new optimization, it would be able to acquire Lk2 even if Thread 1 has not acquired lock Lk1 yet. Basically, when Thread 1 would reach the point where it is about to acquire Lk1, Thread 0 would know that Thread 1 is not going to acquire Lk2 until it would have executed more instructions than what Thread 0 has executed up till now. With this optimization, we can reduce the calculated WCET, albeit at the cost of slightly increased WCET calculation time.

Table II. BENCHMARK CHARACTERISTICS

Benchmark	Basic blocks	Cond branches	Locks	Max L2 cache misses
Fluidanimate (ComputeForcesMT)	11	2	2	12
Network (thread_ippktcheck)	20	11	2	16
Radiosity (radiosity_averaging)	6	3	3	12

Table IV. IMPROVEMENT BY UPDATING CLOCK AHEAD OF A LOCK ACQUISITION FOR RADIOSITY BENCHMARK

Parameters	**2 threads**		**4 threads**	
Configuration	*W/o opt*	*With opt*	*W/o opt*	*With opt*
WCET (cycles)	1440	1378 (**1.04x**)	1728	1516 (**1.15x**)
Calc time (secs)	0.17	0.1 (**1.7x**)	3.33	3.42 (**0.97x**)
Mem consumed (MB)	179.0	178.8 (**1x**)	192.28	201.3 (**0.96x**)
States (millions)	0.021	0.019 (**1.11x**)	0.21	0.34 (**0.62x**)

D. Avoiding binary search

Approaches using model checking to calculate the WCET, such as [9] and [6] use assertions to find the WCET. So, they have to run the model checker several times in a binary mode fashion to reach the right WCET value. Although [6] talks about using the sup operator of UPPAAL to avoid binary search, the current stable release of UPPAAL, which is version 4.0.13, does not support the sup operator. Only versions 4.1 and greater of UPPAAL support the sup operator. That is why, the authors of [6] discuss the sup operator only in the future work section of their paper. On the other hand, our technique of avoiding binary search works perfectly on the stable release version of SPIN.

We avoid performing binary search by logging the value of the elapsed time instead. We make use of the VAR_RANGES flag in SPIN to log the ranges of the variables. However, VAR_RANGES only give ranges from 0-255. Since, SPIN generates C files which are further compiled to make the executable model checking file, it is possible to modify the C code to log the full integer value of the elapsed time. We have written a script that does exactly that by inserting code in the *logval* function to log the value of the elapsed time, the maximum value of which is taken as the WCET. Running the model checker with VAR_RANGES flag increases the calculation time, but is still a much faster method than running several iterations to reach the WCET value through binary search.

IV. PERFORMANCE EVALUATION

In this section, we will discuss the results that we achieved by applying optimizations and using deterministic execution. Section IV-A discuss the results, while Section IV-B shows further improvement achieved by avoiding binary search to reach the WCET value.

A. Results

We selected three benchmarks. One is Fluidanimate from the PARSEC [18] benchmarks suite, one is a Network protocol

Table III. PERFORMANCE RESULTS

Configuration	**Param/BM**	**Fluidanimate (ComputeForcesMT)**		**Network (thread_ippktcheck)**	**Radiosity (radiosity_averaging)**	
Number of cores		**2**	**4**	**2**	**2**	**4**
1. Deterministic with optimized clock updates and hardware support	*WCET (cycles)*	1209	1209	1276	1378	1506
	Calculation time (secs)	0.11	25.1	1050	0.1	3.42
	Memory consumed (MB)	179.8	410.6	11414	178.8	201.3
	States (millions)	0.031	2.9	218.7	0.019	0.34
2. Non-deterministic with hardware support	*WCET (cycles)*	1200 **(0.99x)**	1200 **(0.99x)**	1257 **(0.99x)**	1346 **(0.98x)**	1442 **(0.96x)**
	Calculation time (secs)	0.74 **(6.7x)**	3980 **(158x)**	7300 **(7x)**	0.36 **(3.6x)**	428 **(125x)**
	Memory consumed (MB)	194.2 **(1.08x)**	36595.4 **(89x)**	81919.9 **(7.2x)**	182.3 **(1.02x)**	3445.1 **(17x)**
	States (millions)	0.21 **(6.8x)**	544.4 **(188x)**	1576.6 **(7.2x)**	0.053 **(2.79x)**	46.96 **(138x)**
3. Non-deterministic without hardware support	*WCET (cycles)*	1254 **(1.04x)**	∞	∞	1444 **(1.05x)**	∞
	Calculation time (secs)	7.78 **(71x)**	∞	∞	7.6 **(76x)**	∞
	Memory consumed (MB)	289.5 **(1.61x)**	∞	∞	337.8 **(1.89x)**	∞
	States (millions)	2.03 **(65x)**	∞	∞	1.63 **(86x)**	∞
4. Deterministic without optimized clock updates and with hardware support	*WCET (cycles)*	1224 **(1.01x)**	1364 **(1.13x)**	1297 **(1.02x)**	1509 **(1.1x)**	1889 **(1.25x)**
	Calculation time (secs)	0.46 **(4.18x)**	414 **(16x)**	9450 **(9x)**	0.32 **(3.2x)**	4.49 **(1.31x)**
	Memory consumed (MB)	187.8 **(1.04x)**	3653.3 **(8.9x)**	81919.9 **(7.2x)**	180.8 **(1.01x)**	208.9 **(1.04x)**
	States (millions)	0.13 **(4.19x)**	57.8 **(20x)**	1645.37 **(7.5x)**	0.041 **(2.16x)**	0.47 **(1.38x)**

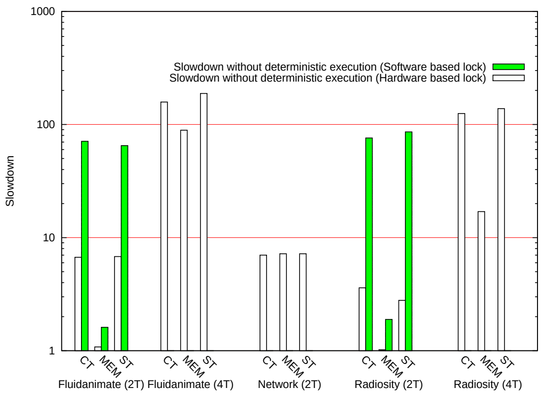

Figure 7. Slowdown with non-deterministic execution w.r.t deterministic execution (Panels 2 & 3 in Table III)

benchmark from EEMBC [2] and lastly we have Radiosity from SPLASH [19]. We only used a portion of these applications. Those portions included shared memory accesses. The characteristics of those parts of the code are shown in Table II. The names of the functions from which the code is taken are also shown. To run these benchmarks, we used a computer with 96GB RAM. We used the DCOLLAPSE flag of PROMELA for compressing memory. The results are shown in Table III.

Using only the approach of [6] and without applying the optimizations discussed in Section III-B, none of the benchmarks could complete, due to the state explosion problem. With our optimizations, most of the configurations could finish. Those configurations that still could not finish are indicated with a ∞ mark in Table III.

The first panel in the Table III shows the results with deterministic execution with clocks updated ahead of time. The second panel shows the results with non-deterministic execution with hardware based lock acquisition, which do not require cache coherence for the shared mutexes. Adding hardware-based lock acquisition is done to have a fair comparison with the deterministic case, because we used hardware based deterministic execution to avoid excessive cache coherency that comes with software based deterministic execution. The third panel shows non-deterministic execution with normal software

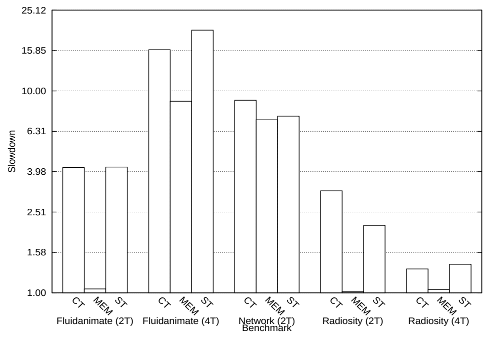

Figure 8. Slowdown of updating clocks after execution (Panel 4 in Table III)

based lock acquisition that does involve cache coherence for the shared mutexes. Lastly, the fourth panel shows the results with deterministic execution that does not update the clock ahead of time but after execution, like Kendo.

From the Table III, we can see that deterministic execution with optimized clock updates (clocks updated ahead of time) gives the best results in terms of calculation time, memory consumed and number of states stored. Introducing deterministic execution does however increase the WCET slightly due to the extra code included in the programs. The increase in WCET however is not more than 4% for the selected benchmarks. The improvement in calculation time, memory consumed and number of states stored scales with the number of threads used. For example, for the Fluidanimate benchmark, the calculation time, memory consumed and number of states were reduced by as much as 158x, 89x and 188x respectively for 4 threads. From Panel 3 of Table III, we can see that the lack of hardware support causes cache coherency for shared mutexes to significantly increase calculation time, memory and states. The comparison of non-deterministic execution with respect to the deterministic version is also illustrated in Figure 7, where the bars on the left (in white color) show the overhead for non-deterministic execution with hardware support while the colored bars on the right show the overhead without hardware support. In cases where the later could not complete, we leave that column empty, that is, no bar is drawn. In that figure, CT represents calculation time, MEM represents memory consumed and ST represents the number of states used

Table V. IMPROVEMENT BY AVOIDING BINARY SEARCH (FOR 4 THREADS)

Benchmark	W/o VR (secs)	With VR (secs)	DE speedup	BS iterations	Overall speedup
Fluidanimate	18.7	25.1	158x	21	**2471x**
Radiosity	2.70	3.42	125x	18	**1775x**

by the model checker.

Our method of updating clocks ahead of time also shows to significantly improve both the WCET and calculation time, as compared to updating clocks after execution. The improvement in WCET happens due to the fact that by updating clocks ahead of time, we reduce the waiting time of a thread waiting for a lock. That waiting also increases the possible number of states, thus increasing the calculation time and memory consumed. The slowdown caused by updating clocks after execution is illustrated in Figure 8.

In Table IV, we discuss the improvement in WCET that we observed by applying the optimization that we discussed in Section III-C, that is, by using the *ahead* and *nl* variables to allow a thread to proceed with lock acquisition even when another thread is waiting for a lock but has a lesser value of logical clock. We show the numbers for the Radiosity benchmark for both 2 and 4 threads. The other two benchmarks do not have different mutexes, so we could not apply this optimization to them. In the *W/o opt* column, we use the basic DetLock mechanism of updating the clock ahead of time, while in *With opt*, we also use *ahead* and *nl* variables. In the *With opt* column, we also show improvement ($>$1x) or degradation ($<$1x) for all the parameters as compared to the *W/o opt* column. From the table, for 4 threads, we can see improvement in WCET at the cost of increased number of states, memory consumption and calculation time. However, these numbers are still much better than the non-deterministic case (see Table III).

B. Improvement by avoiding binary search

In Section III-D, we discussed how we can avoid binary search by modifying the C code generated by SPIN to include the code to log the elapsed time value. This method avoids performing binary search as done by the state of the art approaches that use model checking to calculate WCET, such as [9] and [6]. Table V shows the overall speedup for 4 threads, including that which comes from avoiding the binary search. The column titled *W/o VR* shows the calculation time without using VAR_RANGES (See Section III-D to see discussion about the VAR_RANGES flag), while the *With VR* column shows the calculation time by using it. Next we show the speedup that we achieved with deterministic execution (DE) with clocks updated ahead of time, followed by the number of iterations used to reach WCET if binary search (BS) is used. The overall speedup is then calculated by using the following formula.

$$(Without\_VR/With\_VR) \times DE\_speedup \times BS\_iters$$

From the table, we can see that for the Fluidanimate benchmark, the overall speedup is as high as 2471x.

V. CONCLUSIONS

In this paper, we used model checking for estimating the WCET for portions of the applications where shared memory accesses occurred. We showed that by using deterministic execution, we can reduce calculation time and memory usage significantly at the cost of negligible increase of the calculated WCET. We significantly reduced the time (up to 158x), memory (up to 89x) and states (up to 188x) for calculating WCET with a negligible increase (up to 4%) in the calculated WCET for a multicore system with 4 threads. We also showed an improvement in all the parameters, if we update the deterministic execution clock ahead of time, as in the case of DetLock. Moreover, we avoid performing binary search to calculate the WCET, which involves running several iterations of the model checker, by modifying the C code generated by SPIN to log the value of the elapsed time instead. The total combined gain in speed was found to be as high as 2471x. The state explosion problem still poses a challenge to this solution for practical purposes though. Future work will focus on an approach which combines static analysis with model-checking might be used to overcome that problem.

REFERENCES

[1] http://spinroot.com/spin/whatispin.html

[2] http://www.eembc.org/

[3] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[4] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails? In *2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[5] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, 2010.

[6] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards wcet analysis of multicore architectures using uppaal. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[7] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, March 2009.

[8] H. Mushtaq, Z. Al-Ars, and K. Bertels. Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 721–730, 2012.

[9] Lan Wu and Wei Zhang. A model checking based approach to bounding worst-case execution time for multicore processors. *ACM Trans. Embed. Comput. Syst.*, 11(S2):56:1–56:19, August 2012.

[10] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *RTAS*,2008.

[11] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *RTCSA*, 2009.

[12] Hamid Mushtaq and Zaid Al-Ars and Koen Bertels. Accurate and efficient identification of worst-case execution time for multicore processors: A survey. In *IDT 2013.*

[13] S. Bygde. Static WCET Analysis Based on Abstract Interpretation and Counting of Elements. Lic. dissertation, School of Innovation, Design and Engineering, March 2010.

[14] Xianfeng Li, Yun Liang, Tulika Mitra, Abhik Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. Science of Computer Programming, Special issue on Experimental Software and Toolkit, 69(1-3), December 2007.

[15] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET Analysis of Real-Time Parallel. In *13th International Workshop on Worst-Case Execution Time Analysis*, 2013.

[16] Mike Gerdes, Theo Ungerer and Rudolf Knorr. Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores. Phd Thesis.

[17] Sudipta Chattopadhyay. Time-predictable Execution of Embedded Software on Multi-core Platforms. Phd Thesis.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[19] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[20] J. Gustafsson, B. Lisper, C. Sandberg and N. Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *WORDS*, 2003.

7 CONCLUSIONS AND FUTURE RESEARCH

In this chapter, we first discuss the conclusions of this thesis in Section 7.1 and then give recommendations for future work in Section 7.2.

7.1. CONCLUSIONS

In this thesis, we discussed two different techniques for improving reliability of multicore systems, record/replay and deterministic multithreading. Besides that, we also showed how the calculation time of WCET can be significantly brought down by using deterministic multithreading. Below, we discuss the conclusions that we drew from Chapters 2 to 6.

CHAPTER 2

In this chapter, we gave a survey of different fault tolerance techniques employed for shared memory multicore systems. With the advent of nano-scale technology, the transistor sizes have shrunk a lot. However the smaller transistor sizes are more prone to transient and permanent faults. Fault tolerant systems can employ redundancy to check for errors. We highlighted the difficulties in tolerating faults for multicore systems. The main conclusions are the following.

1. Synchronization operations for shared memory accesses can become a frequent source of non-determinism. This creates problems in fault tolerant systems for multithreaded programs running on multicore systems, as the redundant processes must execute identically in the absence of any fault.

2. To make sure redundant processes execute identically, we can use deterministic multithreading, which can be both language based and runtime based. The problem with the language based techniques though is that they have a steep learning

curve, and it is easy to introduce deadlocks. Therefore, it is preferable to use runtime based methods.

CHAPTER 3

In this chapter, we discussed our own implementation of fault tolerance for shared memory multicore systems using record/replay. We showed how our method is faster, more portable and more scalable than the state of the art approaches. The following conclusions can be drawn from that chapter.

1. Reducing atomic operations for accessing the queue for record/replay can make it more scalable as it reduces the contention between the cores.
2. The queue can be further optimized by eliminating true and false sharing of the cache lines.
3. The cost of error detection in user-space can be reduced by segmenting memory into multiple pages, so that there are less page faults for identifying modified memory.
4. The hardware CRC instruction supported by modern processors can significantly reduce the time for memory comparison using the checksum method.

The overhead incurred by our approach ranges from 0% to 18% for selected benchmarks. This is lower than comparable systems published in literature.

CHAPTER 4

In this chapter, we discussed our own implementation of deterministic multithreading, which like record/replay, can be used for deterministic execution of redundant processes running on multicore systems for fault tolerance. The following conclusions can be drawn from that chapter.

1. Besides being more portable, a compiler based deterministic multithreading approach can outperform an approach which relies on hardware performance counters for logical clocks due to the fact that the compiler based approach can update those logical clocks ahead of time.
2. It is possible to make deterministic multithreading much more efficient if we assume there are no race conditions in the program.
3. Runtime deterministic multithreading would always be limited in scalability due to the requirement of global communication among the threads.

For 4 cores, the average overhead of these clocks on tested benchmarks is brought down from 16% to 2% by applying several optimizations. Moreover, the average overall overhead, including deterministic execution, is 14%. We also employed our deterministic multithreading scheme for fault tolerance.

CHAPTER 5

In this chapter, we compared the performance of record/replay with deterministic multithreading for fault tolerance. Moreover, we created a hardware based version of our deterministic multithreading method (DetLock). The following conclusions can be drawn from that chapter.

1. Record/replay is generally faster. However, the benefit of deterministic multithreading is that it does not require communication among the redundant processes for synchronization operations and this therefore improves its isolability, which is crucial for reliability and fault tolerance.

2. For some benchmarks, our compiler based deterministic multithreading approach combined with our hardware, outperforms fully hardware based deterministic multithreading approaches.

For record/replay, the overhead is less than 25% for all benchmarks down to approximately 0% for some benchmarks. For deterministic multithreading, it depends on the usage of shared memory. With low shared memory usage, the overhead is less than 25% while with high usage, it can reach up to 56%. We also compare our deterministic multithreading scheme (DetLock) to other approaches, for example Kendo, and show overhead reduction up to 189% for 8 threads. Even when Kendo is assisted with hardware, the hardware assisted DetLock version can improve performance up to 26% over Kendo.

CHAPTER 6

In this chapter, we used deterministic multithreading to calculate WCET of a multithreaded real time process. The following conclusions can be drawn from the chapter.

1. With deterministic multithreading, we can significantly reduce the WCET calculation time, as the possible number of states a program can reach is reduced.

2. However, for deterministic multithreading to reduce WCET calculation time, we need a hardware based mechanism for comparing logical clocks. This is needed because software based approaches would increase cache coherence activity and therefore increase the WCET calculation time.

By optimizing the model checking code, we are able to complete benchmarks which otherwise have state explosion problems. Furthermore, by using deterministic execution, we significantly reduce the calculation time (up to 158x), memory (up to 89x) and states needed (up to 188x) for calculating WCET, with a negligible increase (up to 4%) in the calculated WCET for a multicore system with 4 cores. Lastly, unlike other state-of-the-art approaches, that perform binary search to search the WCET by running several iterations, our method calculates WCET in just one iteration. Taking all these optimizations into consideration, the gain in speed of WCET calculation is from 1775x to 2471x for 4 threads.

7.2. FUTURE RESEARCH

The following topics can be considered for future research.

Combining runtime deterministic multithreading with language based deterministic multithreading

In this thesis, we limited our evaluation to 8 cores. In the future, we would like to have an efficient deterministic multithreading method that can work for many cores. As we increase the cores, the overhead increases significantly. Therefore, for manycore systems, we may need to combine language based deterministic multithreading with runtime based multithreading. As discussed in Chapter 2, the disadvantage of using only language based multithreading is the increased programming difficulty. In addition, there is a high probability of introducing deadlocks into the code. By combining runtime method with language based method, it would be possible to have little changes in popular languages, such as C, to ensure deterministic execution, thus making it easier to write multithreaded programs that execute deterministically.

Evaluate our fault tolerant scheme with fault injection

In this thesis, we did not evaluate the fault coverage of our fault tolerant scheme. In the future, we would like to conduct fault injection experiments to do so.

Deterministic multithreading for mission-critical systems

In this thesis, we ran the redundant processes on the same machine. However, for mission critical systems, it makes more sense to use separate machines. In the future, we could evaluate the performance of deterministic execution with multiple machines. Moreover, we used the technique of checkpoint/rollback with double modular redundancy. For mission critical systems, however it makes more sense to use triple modular redundancy with a faster recovery method.

Combining model checking with static analysis for calculating WCET with deterministic multithreading

In this thesis, we used a model checking based approach to calculate WCET. However, the model checking based approach does not scale well even with deterministic multithreading. An approach that combines static analysis with model checking could be used to make it more scalable. With such an approach, it may become possible to calculate WCET for real world programs running on existing multicore processors.

LIST OF PUBLICATIONS

INTERNATIONAL JOURNALS

1. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Efficient and highly portable deterministic multithreading (DetLock)*, Computing, vol. 96, pp. 1131-1147, 2014

2. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Fault Tolerance on Multicore Processors using Deterministic Execution*, Submitted to ACM transactions on Computer Systems.

INTERNATIONAL CONFERENCES

1. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems*, Design and Test Workshop (IDT), 2011 IEEE 6th International, pp. 12-17, 11-14 Dec. 2011

2. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *A user-level library for fault tolerance on shared memory multicore systems*, Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on, pp. 266-269, 18-20 April 2012

3. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *DetLock: Portable and Efficient Deterministic Execution for Shared Memory Multicore Systems," High Performance Computing, Networking, Storage and Analysis (SCC)*, 2012 SC Companion, pp. 721-730, 10-16 Nov. 2012

4. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Efficient software-based fault tolerance approach on multicore platforms*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, pp. 921-926, 18-22 March 2013

5. **Mushtaq, H.**; Al-Ars, Z.; Bertels, K., *Fault tolerance on multicore processors using deterministic multithreading*, Design and Test Symposium (IDT), 2013 8th International, 16-18 Dec. 2013

6. **Mushtaq, H**.; Al-Ars, Z.; Bertels, K., *Accurate and efficient identification of worst-case execution time for multicore processors: A survey*, Design and Test Symposium (IDT), 2013 8th International, 16-18 Dec. 2013

7. **Mushtaq, H.**; Sabeghi, M.; Bertels, K., *A Runtime Profiler: Toward Virtualization of Polymorphic Computing Platforms*, Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on , pp. 144,149, 13-15 Dec. 2010

8. M. Sabeghi, **H. Mushtaq**, K.L.M. Bertels, *Runtime Multitasking Support on Reconfigurable Accelerators*, 1st International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2010), 1 June 2010, Tsukuba, Japan

CURRICULUM VITÆ

Hamid MUSHTAQ

Hamid Mushtaq was born in Karachi, Pakistan in 1980. He received his Bachelors of Electrical Engineering with President Gold Medal (for being best in academics) in 2002. Thereafter, he worked for several different governmental and private companies in Karachi for 5 years, mostly in the field of embedded software. In 2008, he came to Delft to pursue his masters degree in Embedded Systems, which he completed in 2010. The same year, after finishing his masters degree, he started working as a Phd student in the department of Computer Engineering, under the supervision of Dr. Zaid Al-Ars and Prof. Koen Bertels. His research interests include runtime systems, compilers, parallel programming, reliable systems and big data.