EXPLOITING SPECULATIVE AND ASYMMETRIC EXECUTION ON MULTICORE ARCHITECTURES

Jons-Tobias Wamhoff
Born on: 5 September 1981 in Erfurt, Germany

DISSERTATION

to achieve the academic degree

DOCTOR OF PHILOSOPHY (PH.D.)

First referee
Prof. Christof Fetzer, Ph.D.
Technische Universität Dresden, Germany

Second referee
Prof. Pascal Felber, Ph.D.
Université de Neuchâtel, Switzerland

Submitted on: 27 June 2014
Defended on: 21 November 2014
PREFACE

The design of microprocessors is undergoing radical changes that affect the performance and reliability of hardware and will have a high impact on software development. Future systems will depend on a deep collaboration between software and hardware to cope with the current and predicted system design challenges. Instead of higher frequencies, the number of processor cores per chip is growing. Eventually, processors will be composed of cores that run at different speeds or support specialized features to accelerate critical portions of an application. Performance improvements of software will only result from increasing parallelism and introducing asymmetric processing. At the same time, substantial enhancements in the energy efficiency of hardware are required to make use of the increasing transistor density. Unfortunately, the downscaling of transistor size and power will degrade the reliability of the hardware, which must be compensated by software.

In this thesis, we present new algorithms and tools that exploit speculative and asymmetric execution to address the performance and reliability challenges of multicore architectures. Our solutions facilitate both the assimilation of software to the changing hardware properties as well as the adjustment of hardware to the software it executes. We use speculation based on transactional memory to improve the synchronization of multi-threaded applications. We show that shared memory synchronization must not only be scalable to large numbers of cores but also robust such that it can guarantee progress in the presence of hardware faults. Therefore, we streamline transactional memory for a better throughput and add fault tolerance mechanisms with a reduced overhead by speculating optimistically on an error-free execution. If hardware faults are present, they can manifest either in a single event upset or crashes and misbehavior of threads. We address the former by applying transactions to checkpoint and replicate the state such that threads can correct and continue their execution. The latter is tackled by extending the synchronization such that it can tolerate crashes and misbehavior of other threads. We improve the efficiency of transactional memory by enabling a lightweight thread that always wins conflicts and significantly reduces the overheads. Further performance gains are possible by exploiting the asymmetric properties of applications. We introduce an asymmetric instrumentation of transactional code paths to enable applications to adapt to the underlying hardware. With explicit frequency control of individual cores, we show how applications can expose their possibly asymmetric computing demand and dynamically adjust the hardware to make a more efficient usage of the available resources.
ACKNOWLEDGEMENTS

Over the last years, many people were involved in the creation of this thesis. I would like to take the opportunity to thank everybody who supported me, including those that I forgot to mention.

I am very grateful to my supervisor Prof. Christof Fetzer for giving me the opportunity to carry out my doctorate studies in his research group. He brought the relevant problems to my attention that make the foundation of this thesis and guided me with insights how to research and solve these problems. Thank you!

I thank Prof. Pascal Felber, who is coauthor on the majority of my publications, for assisting and teaching me in scientific writing. Besides being a great motivator and driver for my projects, he showed me how to bring papers in a perfect shape.

I also thank my coauthors, Dave Dice, Stephan Diestelhorst, Rasha Faqeh, Patrick Marlier, Gilles Muller, Torvald Riegel, Etienne Rivière and Mario Schwalbe, who were essential partners for my research. Without their collaboration, many of my publications would not have happened. In particular, Rasha Faqeh developed collaboratively with me the transactional memory for state replication and recovery (Section 4.5), Mario Schwalbe worked with me on the transactional encoding (Section 4.4) and furthermore implemented and evaluated our algorithms (Section 4.6), Patrick Marlier carried out the Intel Turbo Boost measurements (Section 6.5), and Stephan Diestelhorst investigated the delegation of critical sections to boosted cores (Section 6.6.4).

Furthermore, I would like to thank my colleagues at the Systems Engineering Group for lively discussions, feedback and motivation throughout the period of my doctoral work. Especially, my office mates Diogo Behrens, Marc Brünink, Sebastian Ertel, Martin Nowack, Torvald Riegel and Stefan Weigert enhanced my working environment with continuous backing and entertainment.

Last but not least, I am thankful to my family for their continuous motivation, support and extensive patience. I love you!
## Contents

1 Introduction  
1.1 Overview of Current and Future Hardware Challenges  
1.1.1 Historic Frequency Scaling  
1.1.2 The Multicore Scaling Era  
1.1.3 Dark Silicon and Heterogeneous Cores  
1.1.4 Reliability of Future Processors  
1.2 Overview of Parallel Software and its Synchronization  
1.2.1 Parallel Computing  
1.2.2 Shared-Memory Synchronization  
1.2.3 Transactional Memory  
1.3 Approaches and Contributions  
1.3.1 Approaches Towards Software Challenges  
1.3.2 Contributions  

2 The Universal Transactional Memory Construction  
2.1 Introduction  
2.2 Related Work  
2.3 Asynchronous Multicore System Model  
2.3.1 Threads and Synchronization Primitives  
2.3.2 Invocations and Transactions  
2.4 Universal Transactional Memory Construction  
2.4.1 Universal Construction for the Good Case  
2.4.2 Dealing with Thread and Transaction Crashes  
2.4.3 Tolerating Non-Terminating Transactions  
2.5 Proof of Correctness  
2.6 Summary  

3 RobUSTM: A Robust Software Transactional Memory  
3.1 Introduction  
3.2 Related Work  
3.3 System Model  
3.4 Design and implementation  
3.4.1 Why a Lock-based Design?  
3.4.2 Optimizing for the Good Case  
3.4.3 Progress and Fairness  
3.4.4 Safe Lock Stealing  
3.4.5 Dealing with Crashed Transactions  
3.4.6 Dealing with Non-Terminating Transactions  
3.5 Evaluation  
3.5.1 Throughput for Well-Behaved Transactions  
3.5.2 Tolerating Crashes and Non-Terminating Transactions  
3.6 Summary  

4 Transactional Encoding for Tolerating Transient Hardware Errors  
4.1 Introduction  
4.2 Related Work  
4.3 System Design  
4.3.1 System Model  
4.3.2 Design Overview  
4.4 Error Detection  
4.4.1 Encoded Processing  
4.4.2 Application Encoding  

ix
Bibliography

Lists of Figures, Tables and Algorithms

List of Figures ........................................ 143
List of Tables ......................................... 145
List of Algorithms ..................................... 147
1 INTRODUCTION
Almost a decade ago, the chip development with ever increasing clock speeds hit the power and memory wall. The transistor density gains that result from the continuation of Moore's Law are since then used for adding an increasing number of cores on a single die. This resulted in a reduced single-threaded performance compared to a uniprocessor that must be outweighed by parallel execution on symmetric multicore processors. For the future, it is predicted that this multicore scaling is limited by power constraints imposed by thermal cooling and power delivery, as well as limited available parallelism in applications. New means are required to improve the energy efficiency of the chip in order to improve the overall computing power. The solutions will include asymmetric general-purpose multicores and customized heterogeneous cores. Asymmetric multicores use the available power envelope to either run all cores at normal speed for a highly parallel execution or a subset of cores at a higher frequency to speed up sequential portions. Heterogeneous cores improve the energy efficiency by specialization, e.g., custom accelerators, but cannot afford to power on all transistors concurrently, which is known as the dark silicon problem. Such processors are also predicted to be in part unreliable because of processing variations with shrinking feature sizes, near-threshold voltage and increased sensitivity. Figure 1.1 gives an overview of the hardware developments over time and the challenges we address in this thesis.

Figure 1.1: Overview of the current and future hardware developments, its implied software challenges and a brief overview of our contributions.

Future systems will depend on a deep collaboration of software and hardware to cope with the performance and reliability challenges. In general, we believe that applications will gain more power over the hardware to control its execution. This allows to dynamically adapt the hardware to the workload characteristics and to process algorithms with asymmetric speeds of cores. The processing itself will demand increasing parallelism to speed up the overall execution, which requires a scalable synchronization. The synchronization of the communication within parallel systems must not only be scalable but also robust such that it can guarantee progress in the presence of hardware faults. With the unreliability of hardware, software must not only guarantee progress but also correctness by becoming fault tolerant.

In this thesis, we will investigate tools and algorithms that help programmers to develop efficient and resilient parallel software by applying robust concurrency control and exposing asymmetric workload distributions. We target efficient parallelism using speculation and optimistic concurrency control and show how to guarantee progress in a speculative environment by helping, lock-stealing or switching to a pessimistic execution. Our performance improvements are based on exposing asymmetry with a lightweight instrumentation for speculative
synchronization and manual control of the frequency of individual cores. We address the unreliability of cores by tolerating hardware faults that may lead to crashed or misbehaving threads, which might interfere with the progress of other correct threads, and tolerate single event upsets using a combination of encoded execution and replicated state.

In this chapter, we first give an overview of the hardware development an the challenges towards software. Then, we provide background information on the parallel computing landscape and finally present our contributions.

1.1 OVERVIEW OF CURRENT AND FUTURE HARDWARE CHALLENGES

The chip development is undergoing dramatic changes due to conflicting dependencies in the scaling of feature sizes, energy efficiency and frequency. This development shifted in the last decade from ever increasing clocks speeds of uniprocessors to symmetric multicore processors with increasing numbers of cores. In the future, the design will soon shift again, towards asymmetric and heterogeneous multicore processors that will eventually not be able to power on all cores simultaneously. At the same time, the reliability of the cores will decrease due to variations during manufacturing or degradation over time.

In this section, we describe the past and future developments hardware is undergoing and derive challenges towards software that is executed on such systems.

1.1.1 HISTORIC FREQUENCY SCALING

For the last decades, chip development has been driven by Moore’s law [Moo65], which states that approximately every two years the complexity of integrated circuits doubles, as shown in Figure 1.2. Each new generation of chips that is developed within that time frame can fit twice the number of transistors mainly because of the shrinking minimal feature size [Boh07]. Until approximately 2005, Moore’s law was accompanied by Dennard scaling (also known as MOSFET scaling) [Den+74], which states that the power density stays constant while the transistor size decreases, or, the transistor’s power usage scales downwards proportional to its linear size. Moore’s law and Dennard scaling allowed to improve the performance per watt exponentially by fitting more and faster transistors on a chip without increasing the overall power consumption.

The switching power dissipation $P$ of a processor is defined as a function of supply voltage $V$, operating frequency $f$ and the dynamic gate capacitance $C$: $P = V^2 \times f \times C$ [HP11].

Dennard scaling, which reflects the historic chip development, states that the transistor’s physical features [AL05], such as supply voltage, channel length and gate oxide thickness, were reduced concurrently with a constant factor that keeps the electric field constant. The reduction of the supply voltage is of high importance because it has a quadratic impact on the power dissipation and is required to keep the electric field constant to maintain reliability. With each generation, the supply voltage is reduced by 30%, which leads to an power reduction of 50%. The capacitance of a single transistor scales with the same factor as the channel length and oxide thickness. The total capacitance of the chip is only indirectly depending on the transistor’s capacitance: If the length of the transistor shrinks by 30% (0.7x) it needs only 50% of the area and allows to double the transistor density, which is in line with Moore’s law. At the same time the frequency increases by 40% (1.4x switching speed) due to the delay reduced by 0.7x with smaller transistors [BC11]. Additionally, the frequency could be increased within the limits of the thermal design power (TDP), which specifies the general power dissipation budget without overheating. These effects allowed to improve single-thread performance with each hardware generation, as shown in Figure 1.3.

Unfortunately, nowadays Dennard scaling is considered broken [Esm+11a]. The main reason is current leakage inside the transistor due to the following problems: The minimal feature
1.1 Overview of Current and Future Hardware Challenges

Figure 1.2: The number of transistors continues to increase according to Moore’s law but the further scaling of the frequency discontinued and is substituted by a scaling of the cores per chip (based on data by CPU DB [Dan+12]).

Figure 1.3: The scaling of the thermal design power is constraint by power delivery and heat dissipation, which requires to improve the energy efficiency by reducing the supply voltage such that the (SPECint) performance can continue to increase (based on data by CPU DB [Dan+12]).
size became so small such that \textit{gate oxide leakage} became a problem that could be fixed temporarily using materials with a higher dielectric constant. The other problem is \textit{sub-threshold leakage} due to an insufficient barrier that increases exponentially with the reduction of the threshold voltage. The sub-threshold leakage reached a substantial part of the overall chip power because the voltage reduction cannot outweigh the static power loss \cite{Boh07}. The current leakage additionally causes the chip to heat up, which in turn further increases the static power loss and can create a threat of thermal runaway. As a result, the threshold voltage cannot be further reduced without making the transistor unreliable. The supply voltage is constrained by the threshold voltage, which prevents a further scaling. With the inability to scale the voltage and an increased static power loss due to current leakage, the TDP left no headroom in the switching power dissipation $P$ to further increase the operating frequency $f$.

The ever increasing frequencies also ran into difficulties to keep the single-core processor tuned for high performance busy. To keep the \textit{instructions per cycle} constant, several microarchitecture techniques were implemented. Inside the processor, instruction-level parallelism (ILP) was introduced to further speed up the execution of the single instruction stream. Solutions include deeper pipelining, branch prediction, out-of-order execution or instruction prefetching \cite{HP11}. While caches and ILP are efficient for many applications, they could not prove as a general solution for the ever increasing frequencies. One of the major problems was the growing gap between the fast scaling of the processor speed and the slower improvements in dynamic memory (DRAM) performance. The focus of DRAM development was on memory density and lower cost. To restrict the performance bottleneck, multi-level cache hierarchies were introduced, as shown in Figure 1.4. The caches increased in size because they are more energy efficient than the energy-intensive logic to implement microarchitecture techniques \cite{BC11}.

![Levels and sizes of on-chip caches](image)

\textbf{Figure 1.4:} The scaling of on-chip cache hierarchy levels and their size is a result of the number of available transistors and the increasing performance gap between the CPU and DRAM (based on data by CPU DB \cite{Dan+12}).
1.1 Overview of Current and Future Hardware Challenges

1.1.2 THE MULTICORE SCALING ERA

The scaling of ever increasing frequencies came to an end with the broken Dennard scaling, which is limited by voltage scaling and power consumption, and difficulties to keep the instructions per cycles constant. With the continuous transistor density gains of Moore’s law, the manufacturers shifted towards multicore designs in order to make use of the increasing number of transistors. Figure 1.2 shows the initiation of multicore scaling when frequency scaling discontinued. Multicore processors provide thread-level parallelism (TLP) and dramatically change the requirements towards software to scale with the increasing number of cores [Sut05].

The frequency of the cores is constraint by a reasonable power consumption of the entire chip. This requires a limitation of the frequency as the transistor density increases and more cores are added [BC11]. The total chip performance shifts from a dominance of the single-thread performance towards using multiple cores to improve the computational throughput and reduce latency.

The software must introduce parallelism in order to benefit from the available transistors and scale in performance. The operating system provides means to actually run a distributed system on the cores, i.e., the software is distributed among the different symmetric cores of the processor and processes a workload in parallel.

Amdahl’s law [Amd67] defines the maximal theoretical speedup using a given number of cores ($n_{processors}$):

$$\text{Speedup} = \frac{1}{r_{\text{sequential}} + \frac{r_{\text{parallel}}}{n_{\text{processors}}}}$$

The speedup is limited by the fraction of execution time that is strictly serial ($r_{\text{sequential}}$) and improves the higher the fraction of parallelized execution gets ($r_{\text{parallel}}$, with $r_{\text{sequential}} + r_{\text{parallel}} = 1$). The maximum speedup converges at $\frac{1}{1 - r_{\text{parallel}}}$. Figure 1.5 shows for selected sequential fractions the achievable speedup. If for example only 5% sequential execution time remains ($r_{\text{sequential}}$), then the maximum achievable speedup is 20, a low number considering the predicted many-core systems [Bor07]. Amdahl’s law is pessimistic in the sense that it assumes a linear speedup with the number of cores and does not include other effects such as caching benefits that can sometimes lead to super linear speedup.

Developing parallel software is costly because it requires expert knowledge and is error prone (see Section 1.2). Additionally, software is not infinitely parallel: A high number of processors leads to a more complex task scheduling and higher latencies due to data movement. I/O operations such as network communication or disk access can also become a bottleneck [Esm+11a]. The tasks that are executed on top of the operating system must typically synchronize. This synchronization can also severely harm parallelism if not done correctly because it increases $r_{\text{sequential}}$.

However, as the number of cores will increase in the future, it is still an open question how the larger numbers of transistors and available features will be assigned to the cores. Even with a uniform instruction set architecture (ISA) the actual implementation can be heterogeneous, e.g., by varying the cache size or supported microarchitecture techniques such as energy costly out-of-order execution. The choices include (1) a smaller number of large cores with an emphasis on single-thread throughput, (2) a larger number of small cores that enforce more parallelism, or (3) a hybrid approach that can selectively support single-thread performance or parallelism [BC11; Kum+03].

**Software Challenge 1.** Since the frequencies are hardly expected to increase, software must expose increasing symmetric parallelism in order to achieve better performance. Therefore, programming models must be applied that exploit data and task-level parallelism. As the main delimiter of speedup, the serial portion of the application must be as small as possible.
1.1.3 DARK SILICON AND HETEROGENEOUS CORES

The multicore scaling is predicted to come to an end because of ultimately limited parallelism in applications and power constraints [Esm+11a], as well as thermal cooling [Bor05]. While the scalability and performance of transistors will be able to improve substantially using new materials or transistor types (e.g., carbon nanotubes), their deployment is far in the future. Until then, other radical approaches are needed to further increase the density of computation and maximize the energy efficiency.

The main impact on the power consumption has the reduction of the supply voltage. With near threshold computing, the supply voltage is reduced from the nominal towards the threshold voltage and is expected to improve the energy efficiency by 10x or more [Dre+10]. Unfortunately, it comes along with a reduction of the frequency because the switching speed of the transistors is proportional to the difference between supply voltage and threshold voltage. This results in 10x or greater performance loss. In the sub-threshold region, the switching delay
increases exponentially, which makes it not a viable option. The voltage scaling is ultimately limited by a near-exponential increase of leakage energy, which is the product of leakage current, supply voltage and delay. A reduction of the frequency hurts the single-threaded performance that must be outweighed by the multiple cores added to the chip, aggressive parallelism and enhanced communication.

Besides adding more cores, the increasing number of transistors was used to add more but specialized functionality to the chip to improve the performance while reducing the energy consumption. Examples include a floating point unit (FPU), larger caches, integrated memory controllers or media processing engines. Most of these features have in common that they consume less power than logic and that they are not active permanently, which further reduces the overall power of the chip.

It is predicted that in the future we cannot afford to power on all cores simultaneously, leaving an increasing fraction of the chip area powered off, which is known as the dark silicon problem [Esm+11a]. As a mid-term solution, the cores will continue to support the same ISA but with an asymmetric feature-set that defines their designated performance, e.g., with or without speculation [Kum+03]. The computational requirements must then dynamically adapt to the hardware specifics: Select the most efficient core for the workload, e.g., many small cores for parallel sections or few powerful cores for sequential areas, and power down all other cores that are currently not needed. It has also been proposed to dynamically combine several small cores to speed up sequential sections, e.g., to perform thread-level speculation or to host helper threads for management tasks or inter-core cache prefetching [HM08; KST12].

The asymmetric cores can dynamically alter their characteristics to meet the energy and performance constraints. One existing solution is dynamic voltage and frequency scaling (DVFS) [HM08], e.g., AMD’s Bulldozer architecture supports to run different cores at different frequencies with distinct voltages [AMD12]. DVFS reduces the power consumption during periods of low CPU usage by switching to a lower performance state. With power gating entire parts of the processor can be powered-off when they are not needed, which further minimizes the power consumption. In fact, the importance of power gating unused components of the chip at a fine granularity and with low latency will become more important. For that reason, Intel incorporated in its Haswell architecture the fully integrated voltage regulator (FIVR) onto the chip [Int14]. In contrast to energy conservation, many DVFS implementations allow to switch to a high performance state that exceeds the base operation state to boost critical regions of an application. This is possible if not all features of the processor are currently used to provide the energy and thermal headroom.

The ARM Big.LITTLE processor applies the idea of entirely asymmetric cores in practice [Gre11]. It combines low-power cores with a limited feature set for low intensity tasks with full-fledged cores that provide the performance for high computing demands. All cores are based on the same ISA and provide a coarse-grained trade-off between power efficiency and performance. Within their performance range, all cores can additionally be adjusted using DVFS: The workload is migrated to the high-performance cores only if the frequency requirements exceed the capabilities of the low-power cores.

On the long term, it is predicted that the chip design will move from general-purpose cores towards customized heterogeneous components [BC11; CSG11]. The idea is that potential energy efficiency and performance gains are only possible by combining specialized architectures: The optimization problem previously attacked by general purpose computing using multiple homogeneous cores is partitioned into smaller discrete subsets. These subsets can be optimized individually using heterogeneous cores that are customized for special tasks. Traditionally, the hardware includes already accelerators as separate chips on board. Some of these accelerators together with new features will be implemented as micro-engines on-chip. While the chip area is not free, the micro-engines are expected to achieve a better performance benefit than just using the available transistors for larger caches. Examples for
heterogeneous cores include complex task management micro-engines, which allow to keep computing cores lightweight through a centralized synchronization and delegation, image and media processing cores or micro-engines optimized for scalar computing.

Besides the cores itself, all other parts of the processor chip must be optimized for efficiency as well. A large fraction of power is used for moving data around the chip and through the memory hierarchy [BC11]. Data movement can be minimized by maximizing data locality and larger register files. Enhancing the hardware design with 3D integration by stacking multiple layers of silicon reduces the global interconnect and latency using vertical interconnects [Fic+12]. The network-on-chip, as part of the platform, must be optimized with enhanced routing and switching implementations, e.g., using a combined packet and circuit switching.

A practical arising example is AMD’s Heterogeneous Systems Architecture (HSA) [Kyr12]. The upcoming AMD Kaveri accelerated processing unit (APU) will combine a GPU for highly data parallel workloads with a multicore CPU for serial and task parallel workloads [Bou+14]. It also embeds a small ARM core for the creation of secure execution environments. The architecture will also support unified memory (NUMA) that makes the copying of data between GPU and CPU obsolete and reduces the latency. In addition to the unified virtual memory, even on disk, the cache coherence protocol is extended such that GPU and CPU can access each others caches. These features make the shift of parallel workloads between the heterogeneous components more efficient.

Software Challenge 4. Systems will shift more control to the software as optimal solutions cannot be solved transparently by the operating system as in the past. The heterogeneous hardware landscape breaks the abstractions for generic optimizations of applications, e.g., due to customized accelerators. Thus, the operating system must enhance to cope with the heterogeneous hardware and provide means for the software to control it. Besides symmetric parallelism, software must also be able to expose its workload characteristics to the hardware in order to tune the performance of the system, e.g., by manual DVFS control.

Software Challenge 5. A tight partnership between hardware and software is required to solve the energy-proportional computing problem [BC11; CSG11]. The software tool chain must be aware of the micro-engine back-ends. Thus, compilers must generate specific code and the developers must make use of specialized libraries that operate on the heterogeneous cores, e.g., to delegate tasks to the optimal micro-engine or to provide optimized algorithms.

Software Challenge 6. The scheduling must become heterogeneous aware and allow to dynamically adapt to the underlying hardware. This will be combined with advanced auto-tuning strategies as not all hardware and workload combinations are known beforehand. The large number of cores requires fine-grained partitioning, which should be done gradually automated by the compiler for convenience, to allow a better workload distribution. The adaption of the workload to the heterogeneous landscape at runtime can use virtualization with the benefit of a better portability.

Software Challenge 7. The data movement must be optimized by a task scheduling that is aware of data locality, beyond the support of non-uniform memory access (NUMA). As the hardware might drop the support for the unified virtual address space or the global cache coherence protocol, more control for data movement will shift to software, e.g., by remote memory access programming. This will enable a software-centric optimization of the entire memory hierarchy including the network-on-chip.

1.1.4 RELIABILITY OF FUTURE PROCESSORS

The increased transistor density in integrated circuits leads to less reliable hardware and a higher likelihood for transient errors [Bor05]. While the feature size decreases, the sensitivity
to sub-threshold voltage and radiation increases. This is caused by the need to further reduce the supply voltage in order to reduce the energy consumption. With near-threshold computing, a supply voltage close to the transistor’s threshold increases the energy efficiency by an order of magnitude [BC11; Dre+10]. Unfortunately, it is accompanied by a 5\times increase in performance variation due to process variations and 5 orders of magnitude increase in failure rate of memory and logic due to variations in process, temperature and voltage.

The number of correctly operating transistors per die will degrade over time, e.g., due to oxide wear-out. During chip manufacturing, lithography limitations and random process variations affect all transistor dimensions, e.g., channel length, width, junction depths or gate oxide thickness. The variations will make a greater percentage of the overall transistor size as the transistor shrinks. This leads to extreme device variations for the threshold voltage and a higher probability for single event upsets such as bit flips.

A few solutions to cope with the reliability of cores have been proposed by research. The manufacturers could choose to add spare cores to the chip and disable broken cores that are not usable due to variation. This idea can be extended towards a dynamic reconfiguration of the cores depending on their performance capabilities measured during production [BC11]. Their maximum frequency is then bound to a reliable level. Alternatively, one can do a cherry-picking among the redundant cores [Rag+13b]: Assuming a large number of homogeneous cores, optimal subsets can be chosen depending on their designated task, e.g., a smaller subset of cores that proved to withstand high frequencies for serial sections and a larger subset of slower cores for parallel work. The active subset is chosen depending on the application characteristics and all other cores are powered down. This allows to exploit the core-to-core variations due to sub-threshold leakage and clock frequency.

Software Challenge 8. Apart from the dependability mechanisms implemented in hardware, such as redundancy, parity information and error correcting codes, resilience must spawn into the software. The effort needed in software for fault tolerance depends on the hardware features, e.g., small cores will at most implement parity information. Thus, a hardware-software co-design is required that allows the system to adapt to the heterogeneous reliability conditions. Examples for software challenges range from correcting bit flips undetected by the hardware to multi-threaded applications being able to tolerate crashes of individual threads.

Software Challenge 9. The synchronization of highly parallel applications denotes a special challenge because of its global impact on the application. Threads might start a synchronized operation but never finish it, e.g., because they crashed or entered an infinite loop. Thus, the synchronization must tolerate failures of threads such that they do not hinder the progress of other threads indefinitely. Hardware faults must also not impair significantly on the latency of the synchronization, which could in turn result in timeout failures.

1.2 OVERVIEW OF PARALLEL SOFTWARE AND ITS SYNCHRONIZATION

The predictions of future hardware require changes in the development of parallel software. This development is driven by continuous demand for parallelism because the number of processor cores will increase. The applications can improve their performance only if they provide a speedup according to Amdahl’s law that scales with the number of cores, independently if the cores are symmetric, asymmetric or heterogeneous. At the same time, the predicted unreliability of hardware requires the software to tolerate faults, especially when software interacts and communicates among different cores.

To meet the hardware demands, parallel software development is confronted with several challenges: (1) The workload must be partitioned for a highly parallel execution. (2) The partitioning must support a dynamic adaption and possibly unbalanced distribution to the
cores at runtime. (3) The synchronization of the communication between the partitions must be scalable and efficient such that the serial fraction of execution time can be minimized to support further speedup. (4) The software must also be resilient against hardware faults, e.g., by a redistribution of the workload. (5) Failed portions of the workload must not prevent the system from making progress, e.g., because it got stopped by an incomplete synchronization.

In this section, we will present background knowledge on the parallel computing landscape as well as properties and implementations of shared memory synchronization, which will be used throughout the thesis.

1.2.1 PARALLEL COMPUTING

Parallel computing is a computing paradigm that executes many instruction streams simultaneously on multiple cores. The general principle is to solve a large problem concurrently by splitting it in smaller tasks. In practice, these tasks get assigned to the processor cores for execution in parallel. The basic entity of work that the operating system can schedule are processes. The process can spawn extra threads, which are a lightweight subset of processes.

In order to enable parallel computing, the design of an application must incorporate a decomposition of the computation into tasks. This can be done explicitly using processes and threads provided by the operating system or using sophisticated parallel programming models [McK14]. Explicitly programming with threads gives great control over the execution but requires a manual management of the workload distribution among the cores. Parallel programming models create and manage threads implicitly and hide the complexity from the developer. For convenience, they can be based on compiler support to generate tasks or on runtime systems for an automatic workload distribution. While threads are scheduled by the operating system, runtime systems can schedule tasks onto threads. In the context of asymmetric and heterogeneous cores, the decoupling of tasks from threads is very important to be able to adapt to the underlying hardware. The created tasks must have a granularity that is small enough to scale to larger numbers of cores and to allow a rebalancing of the workload if it is processed asymmetrically by the cores or the tasks have different sizes. If the tasks are too small, the management overhead increases and reduces the overall performance and scalability.

A large variety of parallel programming models has been proposed by research and industry [CN10; PBF10]. The parallel programming models support developers in exploiting parallelism automatically and transparently. Examples include OpenMP [DM98], which originally exploited loop-level parallelism automatically based on code annotations, Intel Threading Building Blocks [Rei07] that provides a template library with popular parallel algorithms or Cilk [Blu+95] with its work-stealing task scheduler.

The lifetime of the threads, managed either explicitly or implicitly, differs depending on the threading pattern. The fork-join pattern spawns threads for parallel regions and assigns specific tasks to the threads. After the parallel region, the application continues to execute single threaded. Spawning threads introduces overhead for the operating system because thread control structures must be created. The thread pool pattern minimizes the thread creation overhead by expanding the lifetime of threads to the lifetime of the application. An additional runtime system then schedules the tasks onto the application-level threads.

Multiple threads of an application typically communicate with each other using shared memory because threads share the same address space of the process they belong to. This is emphasized by the memory hierarchy in between the memory and the processor cores that consists of coherent caches [Dre07]. If multiple processes are selected to implement a concurrent system, they communicate typically explicitly across distinct contexts using pipelining or message passing because each process runs in a separate address space.

For the remainder of this thesis, we focus on multi-threaded user space applications with task parallelism that communicate via shared memory. In contrast to embarrassingly par-
Parallel algorithms based on data parallelism, task parallelism is more challenging because the developer or the tool chain must identify which code can be executed concurrently. While this is even more difficult for irregular parallelism [Vis14], the future hardware predictions with asymmetric and heterogeneous cores opens up new possibilities to adapt to these workloads. Application-level controlled asymmetry can be used to improve the performance. We do not assume any specific parallel programming model but make our solutions generally applicable at the interface of the operating system for applications, i.e., explicit threads.

In the presence of faults, threads and processes can crash or behave incorrectly. We focus on non-malicious faults caused by hardware or operational faults that result in transient or permanent errors [Avi+04]. Dealing with thread crashes within one process requires an dynamic workload assignment. Static assignments of task to threads that crash might result in an incomplete computation.

1.2.2 SHARED-MEMORY SYNCHRONIZATION

Parallel applications based on multi-threading that interact using shared memory communication require synchronization to coordinate the access to shared state [HS08]. A sequence of instructions that operate on shared state is called a critical section. Entering a critical section must be mutually exclusive, i.e., one must prevent that two concurrent threads enter the critical section at the same time. Otherwise, an ordered exclusive access to the shared state cannot be guaranteed, which results in a race condition. Race conditions are a violation of the program correctness can lead to state corruption or undefined behavior.

The shared memory synchronization can be implemented by blocking or non-blocking mutual exclusion. Blocking a thread can be achieved by locks that either halt the execution by descheduling using the operating system or spin on a lock variable until the thread can enter the critical section exclusively. Non-blocking synchronization does not postpone the execution of threads, even if they conflict on the same critical section. Non-blocking code typically relies on atomic read-modify-write primitives that are implemented in hardware and provided by the processor. Atomic primitives perform a small sequence of operations as one indivisible step and make no intermediate results visible to other threads. A classic example is the atomic fetch-and-add instruction, which adds a value to a variable and returns its immediate preceding value.

We assume that multi-threaded applications execute non-deterministic. The program does not depend on any deterministic order of entering critical sections with interleavings that must be enforced by the synchronization. The order solely depends on the scheduling of threads and their possibly asymmetric speed.

The synchronization lets threads interfere with each other when they try to enter a conflicting critical section. With blocking synchronization, a thread that gets delayed unexpectedly half-way through the critical section can prevent other threads from making progress. While non-blocking synchronization cannot stop remote threads, it still does not necessarily guarantee that the outcome of the non-blocking operation allows the thread to make progress. Progress conditions capture this behavior for concurrent algorithms and express guarantees about the completion of thread's executions under various conditions. Recently, a unified explanation was published that puts the progress conditions independently of their blocking or non-blocking nature in relation [HS08; HS11].

The progress conditions and their dependencies are illustrated in Figure 1.6. The reasoning about progress is done in terms of abstract method calls that make an algorithm, where a completed method call means that an invocation has a matching response. On an abstract level, minimal progress means that some method calls were completed and maximal progress means that all method calls were completed. Obviously, maximal progress is desired such that all threads can complete their execution of the algorithm, even in the presence of faults.

The progress condition can also be depending on the environment of the threads, namely
the operating system scheduler. The blocking progress guarantees depend on a *fair scheduler* that assigns to each thread an unlimited number of cycles for execution. Non-blocking progress can depend on a *uniformly isolating scheduler* that allows to run a thread in isolation for a duration long enough to make progress or it is *independent* and makes no assumptions about the scheduling. Besides the fair scheduling, blocking synchronization can guarantee progress only if the threads eventually leave the critical section to limit the delay of other threads.

Based on the above characterization, the following progress conditions can be informally specified: Assuming a fair scheduling of threads, *deadlock-freedom* guarantees minimal progress and *starvation-freedom* maximal progress. If the scheduler is uniformly isolating, *obstruction-freedom* guarantees maximal progress. This requires non-blocking synchronization such that descheduled threads cannot prevent progress of the single active thread. *Wait-freedom* is the strongest progress condition and guarantees maximal progress independently of the scheduling. Every thread must be able to complete its method calls within a finite number of cycles. *Lock-freedom*, while also strong, guarantees only minimal progress under the same conditions, meaning that at least some thread makes progress.

With the predicted unreliability of future hardware, a robust synchronization is required. Threads can invoke abstract method calls but might never reach the corresponding response, e.g., a hardware fault can manifest in a crash in the middle of a critical section or in a corrupt control flow. Non-blocking synchronization can tolerate such faults naturally because the correct threads cannot be blocked by incorrect threads. Therefore, non-blocking features are employed if systems must be safe against asynchronous signals or termination [Mic13]. In general, maximal progress gains importance because it can guarantee the liveness of applications in the context of unreliable hardware and its interference with synchronization.

Many algorithms are based on blocking synchronization or provide only minimal progress because they achieve a better performance or are easier to program. Fortunately, means exist that allow to cross the lines in Figure 1.6 towards maximal progress and non-blocking independent synchronization. A common assumption is that the operating system provides a fair scheduler per se, which allows to focus in the implementation for progress on the algorithm and its synchronization. On application-level, the schedule can become uniformly isolating by introducing a *contention manager* that decides upon a conflict which thread can proceed and which has to wait [Her+03]. The contention manager can employ a back-off strategy that delays threads such that one thread can run in isolation, effectively modifying the behavior of the
1.2 Overview of Parallel Software and its Synchronization

Operating system scheduler. Blocking algorithms can implement mechanisms for lock stealing, e.g., after a timeout, such that remote threads are forced to leave a critical section [Wam+10]. That way, descheduled or delayed threads cannot block other threads infinitely. A benevolent scheduler can make deadlock-free algorithms starvation-free by assuming a fair random scheduler and a deadlock-free spin lock algorithm [HS11]. As soon as the lock is released, a randomly scheduled thread can acquire it, ultimately giving all threads the chance to make progress. Helping can transform a lock-free into a wait-free algorithm, e.g., by announcing method calls using consensus objects in a universal construction [Her91].

While wait-free progress is independent and robust, it usually comes at the high cost of communication overhead, e.g., due to helping mechanisms. The synchronization of shared state must be efficient because it would otherwise limit the speedup that we need to scale to large numbers of cores. The design space is very large and ranges from the granularity of critical sections and their assignment to locks to data locality awareness to reduce the cache coherence traffic.

In this thesis, our investigations are twofold: We will explore means to make the synchronization robust against hardware faults and we will exploit asymmetric computation to speed up the synchronization.

1.2.3 TRANSACTIONAL MEMORY

The classic synchronization using locks is impaired with several drawbacks. Coarse granular locking is easy to program but does not scale to large numbers of cores. Fine granular locking enables scalability but comes at the expense of a higher complexity and development cost and is difficult to apply to irregular structures that cannot be partitioned easily. The selected locking scheme relies on conventions that specify which lock protects which shared state or object. The locking scheme is hopefully documented because locks are an implementation mechanism and cannot express the mapping to critical sections on the programming language-level. Programmers must follow the convention, otherwise the concurrent state accesses are not synchronized correctly, e.g., if the order of acquiring locks is not in compliance with the convention, it can easily lead to deadlocks. Overall, this makes explicit programming with locks error-prone and their mapping hard to exchange later for tuning reasons. Once a high effort was taken to realize a custom locking, e.g., in a component or library, developers might want to reuse that code to build larger systems. Unfortunately, locks do not easily compose: The reused components must expose their locking scheme in order to allow a compound synchronization, which breaks information hiding. Then there is no guarantee that the lock acquisition order of the components matches, i.e., if they are compatible with the system-wide order.

Transactional memory (TM) aims to simplify the synchronization of concurrent programming [HLR10]. It provides developers the ability to declare which operations must be synchronized on the language-level without the need to worry about how the synchronization is actually implemented. The example in Algorithm 1.1 shows a transaction that moves money between two bank accounts. The compiler will instrument all accesses to shared memory in this sequence of code such that it is compatible with a runtime system that implements the synchronization. The runtime system ensures that the sequence of instructions within the boundaries of the transaction will be executed atomically and in isolation, i.e., no inconsistent intermediate results can interfere with other threads. All that happens transparently to the developer and supports composability by the nesting of transactions.

The runtime system with the TM implementation can choose to execute transactions concurrently, e.g., by running speculatively using optimistic concurrency control. If in such a setup two transactions conflict with each other, one will be aborted and restarted from the beginning. The TM implementation offers a large design space with many possibilities for optimizations. Since the runtime system is exchangeable, the most efficient implementation
can be selected statically at compile time or even dynamically at runtime.

Within the last ∼20 years since TM has been proposed [HM93], it opened a new research field and attracted a lot of attention. The focus was first on software transactional memory (STM) [ST95], which is implemented entirely in software and generic enough to run on most hardware. Dynamic STM [Her+03; HF03] eliminated the requirement to specify upfront the memory locations that will be accessed by the transaction such that pointer based data structures can be supported. The performance could be improved by shifting from non-blocking progress guarantees to basing the TM implementation on internal locks to synchronize updates to the shared state [DSS06; Enn06]. Combined with a time-based validation of reads visible only to the local transaction [Fel+10; FFR08; RFF07], STM became more efficient and scalable to large numbers of threads [Dra+11]. The latest manifestation of TM as a viable synchronization mechanism culminated in the availability of hardware transactional memory (HTM) in current microprocessors [Cai+13; JSG12; Yoo+13]. Along the line of ongoing research, TM is currently in the process of being standardized as a synchronization construct in C++ [LW14].

In almost all cases, HTM performs better compared to STM. However, the design and implementation in hardware showed to be very complex, which resulted in several limitations. The HTM capacity is limited, i.e., the number of memory locations that can be accessed within a hardware transaction, and HTM support is not available on all platforms. For efficiency reasons in cost and complexity, the hardware support follows a hybrid approach [Dic+09]: The hardware component provides only best-effort guarantees and is used to speed up the performance [Chr+10]. It is combined with a software component that provides a fallback solution if the hardware transaction was not successful, while still remaining scalable and well performing [Dal+11; Rie+11]. Therefore, we will focus on STM as a constant companion of HTM. STM also allows to focus on specific aspects and problems that are not general enough to be implemented in hardware yet.

This thesis investigates how STM can support developers transparently with a robust synchronization that deals with the predicted future hardware challenges. The goal is to achieve maximal progress in the presence of hardware faults, even if the STM follows a lock based implementation. Despite the synchronization, we will also investigate STM as a mean to replicate and recover state.

1.3 APPROACHES AND CONTRIBUTIONS

We follow a divide and conquer approach to address the challenges towards software that arise from the development of future hardware. Our solutions improve the resilience and efficiency in parallel computing on multicore architectures using speculation with STM and asymmetric execution.

In this section, we first present an high-level overview that is based on the common approaches that were taken had how they relate to the addressed software challenges. Then, we give a detailed outline of our main contributions.
We use the following aspects of computer systems to classify the approaches and our contributions:

- **Resilience**: The increased sensitivity of hardware due to further scaling will make it less reliable. Resilience deals with unreliable hardware such that systems are given the ability to continue to deliver a correct service in the presence of non-malicious faults. Despite hardware-implemented dependability mechanism, software is required to become resilient since not all faults can be covered by hardware and will become visible in software.
  - **Fault tolerance**: Hardware faults may affect threads such that they crash, change their control flow or corrupt their state. Fault tolerance enables software with error detection and recovery to mask faults and prevent a failure of the system [Avi+04]. It uses error detection to identify an incorrect behavior or state and recovery to return the system to a valid state or to perform a reconfiguration such that it can continue its execution.
  - **Robust synchronization**: In parallel computing with shared memory, the synchronization is elementary for communication. The synchronization must not hinder threads that are communicating with each other, i.e., threads must not only tolerate faults of the local core but also of cores that run threads they are interacting with. A robust synchronization is fair by guaranteeing progress among threads while tolerating hardware faults that can lead to a failure of individual threads, in particular during communication.

- **Efficiency**: The limitations of symmetric multicore scaling will shift the design towards asymmetric and heterogeneous cores. This puts a new burden on parallel software development as the performance can only be improved by exposing these hardware features to applications. The efficiency is targeted by making optimal use of the cores at runtime.
  - **Dynamic adaption**: The asymmetric speed of cores requires to adjust the workload distribution at runtime to the underlying hardware. Besides the dynamic assignment of tasks to threads and cores that run at different speeds, the dynamic adaption also covers the dynamic tuning of algorithms to hardware. The adaption also implies to gain control over the hardware by software to configure it dynamically according to the workload.
  - **Latency**: The delay of an operation has a large impact on the performance of a system, especially if other parts of a parallel application are depending on that operation. The latency can be improved by reducing the overhead of the operation or its synchronization and by increasing data locality. Additionally, reducing sequential bottlenecks helps the speedup of symmetric parallelism according to Amdahl’s law because it reduces the serial fraction.
  - **Throughput**: The necessity of a high level of parallelism will continue, with scalability to large numbers of cores remaining the mayor issue. Thus, a higher throughput demands an efficient synchronization, possibly with awareness of asymmetric speeds. Performance improvements can be achieved by an optimal scheduling of tasks onto asymmetric or heterogeneous cores depending on their computing requirements.

The chapters of this thesis are largely based on publications that have been modified or extended to fit in the context of this thesis. The following overviews of the taken approaches and our contributions serve as an index to find specific solutions to the challenges.
1.3.1 APPROACHES TOWARDS SOFTWARE CHALLENGES

Here, we show how the common approaches used by our solutions support the software challenges of future hardware. Figure 1.7 gives an overview of the approaches, which are classified by the top-level aspects resilience and efficiency.

We derive four major goals from the software challenges described in Section 1.1: (1) the separation of workloads from traditional operating system threads is required for fault tolerance and asymmetric scheduling, (2) the progress guarantee of execution is required for fault tolerance and a robust synchronization, (3) the performance improvements by asymmetric processing result from increasing parallelism and an efficient synchronization, and (4) the control of software execution and hardware is crucial to achieve an adaption of hardware to the software, the software's tuning and fault tolerance.

Efficiency can be improved by control of the execution and hardware, an increased performance and separating workloads from threads. The improvements result from asymmetry, which is either introduced by the hardware or by the software itself, and synchronization that adapts to the environment for better use of parallelism. We apply the following approaches for efficiency:

- We use helping in algorithms to naturally support the separation of workloads from threads such that the workload can be reassigned to threads that are idle (applied in Chapter 2).
- With isolating techniques, we separate threads, which are suspected to being crashed
or misbehaving for checking, from the other correct threads such that they do not impair their performance (applied in Chapters 2 and 3).

- **Stealing** improves the performance because we can bound the latency of synchronized operations, e.g., when waiting for a lock (applied in Chapters 3 and 5).

- By **speculating** using transactional memory, we implement a fast path for the common good case with reduced detection mechanisms as well as a lightweight synchronization (applied in Chapters 2, 3, 4 and 5).

- The **unbalancing** of execution speeds improves the performance using asymmetric code paths with either a lightweight or heavyweight instrumentation of threads and an asymmetric hardware testbed with distinct frequencies of cores, e.g., depending on the priority. The redistribution of the workload by separation on application-level allows us to use the computing resources most efficiently (applied in Chapters 5 and 6).

- The **boosting** facilities of modern processors allow us to control the frequency of cores by software such that the hardware can be adapted dynamically to application properties for a better performance (applied in Chapter 6).

**Resilience** can be achieved by guaranteeing progress, separating workloads from threads and the control of execution. Hardware faults and software bugs can result in crashed and misbehaving threads. The guarantee of progress ensures that all correct threads can complete their tasks in such an environment. The separation of workloads from threads allows the correct threads to take over tasks that were assigned to crashed threads such that the workload can be completed. The control of execution enables a thread itself to detect and correct its misbehavior. We apply the following approaches for resilience:

- With **helping**, we design a universal construction based on transactional memory that guarantees maximal non-blocking progress (applied in Chapter 2).

- **Isolating** allows us to bound the number of retries of transactions when the progress is at stake (applied in Chapters 2 and 3).

- With **stealing**, we implement a transactional memory with maximal blocking progress and allow a pessimistic execution to always win over a speculative execution (applied in Chapters 3 and 5).

- By **speculating**, we can roll the execution of a thread back in the occurrence of a conflict or detection of an error, e.g., caused by a hardware fault (applied in Chapters 2, 3, 4 and 5).

- With **encoding**, we control the execution of an application for hardware faults and combined with a state replication we guarantee progress by forward recovery (applied in Chapter 4).

### 1.3.2 CONTRIBUTIONS

We now give a detailed overview of our contributions, which are summarized in Table 1.1. The contributions are organized by the low-level aspects fault tolerance, robust synchronization, dynamic adaption, latency and throughput.

We tackle **fault tolerance** for transient and permanent hardware errors. Since threads can fail, a fixed assignment of workload partitions is prohibitive. Instead, we separate the workload from threads, e.g., at the granularity of transactions, and assign parts of the workload dynamically to a thread pool for execution. If threads fail, the workload gets redistributed


<table>
<thead>
<tr>
<th>Aspect</th>
<th>Contribution (Section)</th>
<th>Software Challenge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resilience of unreliable hardware</td>
<td>Decouple workload from operating system threads to tolerate thread crashes ((2.4.2, 3.4.5))</td>
<td>8. Fault tolerant software</td>
</tr>
<tr>
<td></td>
<td>Isolate execution to protect correctly executing code from hardware faults ((2.4.3, 3.4.6))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Detection and masking of transient errors using arithmetic codes and state replication ((4.3))</td>
<td></td>
</tr>
<tr>
<td>Robust Synchronization</td>
<td>Universal construction for maximal non-blocking progress based on helping ((2.4))</td>
<td>9. Robust synchroniztion</td>
</tr>
<tr>
<td></td>
<td>Safe lock stealing for efficient maximal blocking progress ((3.4.4))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pessimistic execution for guaranteed progress in a speculative environment ((5.3.3))</td>
<td></td>
</tr>
<tr>
<td>Efficiency and performance for asymmetric parallel hardware</td>
<td>Dynamically assign transactions that wrap application code to available threads ((2.4.1))</td>
<td>4. Hardware control; 5. Hardware adaptation; 6. Scheduling &amp; Auto tuning</td>
</tr>
<tr>
<td></td>
<td>Dynamically select code path optimal for underlying hardware ((5.3.2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testbed for exposing applications to asymmetric cores ((6.4))</td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>Bound number of retires for fairness during speculative execution ((3.4.3))</td>
<td>1. Symmetric parallelism; 2. Efficient synchronization; 7. Data movement</td>
</tr>
<tr>
<td></td>
<td>Streamlined execution path with significant STM overhead reduction compared to sequential execution ((5.3.3))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Boost execution of serial fractions ((6.6.1))</td>
<td></td>
</tr>
<tr>
<td>Throughput</td>
<td>Lightweight synchronization for partitions that are processed by subsets of possibly asymmetric cores ((5.4, 6.6.2))</td>
<td>2. Efficient synchronization; 3. Asymmetric parallelism; 6. Scheduling</td>
</tr>
<tr>
<td></td>
<td>Improve performance by application controlled DVFS and asymmetric frequencies ((6.6))</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Overview of the aspects and our contributions that address software challenges of future hardware (listed throughout Section 1.1).

such that the remaining threads will guarantee the completeness of the processing. The same is true for non-terminating transactions, i.e., transactions that never commit and release their resources. We isolate the execution of transactions that have difficulties reaching their commit such that they do not interfere with the progress of well-behaved threads and transactions. Tolerating crashes and non-terminating transactions requires to tolerate ill-behavior of other threads based on symptoms such as signals or timeouts. Transient errors, in contrast, might not trigger such symptoms and a thread itself must check its correctness. Traditional approaches add redundancy on different levels, usually parity information and replicated execution. For transient errors we follow a software-only approach that combines arithmetic codes for error detection with STM for state replication and recovery.

We first address **robust synchronization** by a universal construction that converts a sequential object into a concurrent object with maximal progress guarantees. We investigate if this construction can provide wait-free progress under realistic assumptions, i.e., how we can maximize non-blocking progress. Non-blocking synchronization naturally tolerates ill-behavior of other threads because they cannot block its execution. However, lock-based implementa-
1.3 Approaches and Contributions

Transactions are used widely because they are easier to reason and suffer from less indirections or copy operations, e.g., due to helping. Therefore, we then show how to maximize blocking progress in the presence of crashes and non-terminating transactions by enabling safe lock stealing such that threads cannot block others infinitely. Our approach is efficient because we apply robustness mechanism only when the progress is at stake. Reasoning about progress is inherently difficult in STM systems that are implemented speculatively because transactions can abort and retry. We can use pessimistic execution within a speculative environment to ensure progress, i.e., transactions that can never abort and always commit.

The dynamic adaption to future hardware designs also benefits from a decoupling of the workload from the threads and cores. In this context, the goal is not fault tolerance but a balanced asymmetric processing. We present a model that captures all code in transactions and assigns them to threads dynamically such that faster cores can process a larger share of the workload, which is a good fit for task based parallelism. Depending on the underlying hardware configuration, optimized algorithms can improve the performance. We support multiple code paths with different instrumentation and STM implementations that are optimized for different numbers of cores. The code path can be switched at runtime depending on the number of currently available cores. We provide a testbed with manual control of the asymmetric features of current hardware supporting DVFS. It can be used to evaluate algorithms dedicated for future hardware and allows a dynamic control of frequency scaling and power gating.

Our latency improvements are based on the assumption that the delay introduced by the hardware and operating system is bounded because each thread gets eventually sufficient time on a core to execute. Thus, our goal is to minimize the delay introduced on the application-level. One must assume that this delay is initially unbound due to either a continuously aborting speculative execution or hardware faults that hamper the synchronization. While fault tolerance resolves the latter, we improve the fairness among threads, ultimately limiting the number of retries for a guaranteed execution of operations within a bounded number of steps. STM is typically impaired with high overhead due to instrumentation and bookkeeping, which degrades the latency. We address this issue with a streamlined execution path that significantly reduces the overhead of STM compared to an uninstrumented sequential execution. It is built on a lightweight instrumentation and minimal bookkeeping for updates of shared state that additionally benefits from data locality. Using manual DVFS control, we further reduce the latency, e.g., by boosting the speed of sequential bottlenecks.

We enhance the throughput using a lightweight synchronization, which is in particular beneficial for workloads with distinct partitions that are processed by subsets of cores. Subsets with asymmetric cores can further emphasize throughput within the synchronization: Fast cores get only impaired with a lightweight synchronization overhead and the much larger number of smaller cores assist in providing the parallel backend. That way, the throughput of partitions gets a head start compared to other STM algorithms that allow it to perform better on few numbers of cores. Additionally, we improve the throughput by exposing application knowledge for software controlled DVFS and asymmetric frequencies. The assignment of frequencies to threads can be based on priorities or heterogeneous computing demands.
2 THE UNIVERSAL TRANSACTIONAL MEMORY CONSTRUCTION*

*The contents of this chapter first appeared at TRANSACT ’11 [WF11].
2.1 Introduction

The universal construction shows how to convert a sequential algorithm into a concurrent wait-free algorithm. We introduce a variant of this construction that (1) keeps a bounded state, (2) provides wait-free parallel processing, (3) tolerates thread crashes, and (4) handles non-terminating operations. The foundation of this construction is a wait-free transactional memory that is capable of isolating crash failures and non-termination failures.

We are particularly interested in the question if the introduction of parallelism into an application can facilitate its fault tolerance when executing on unreliable hardware. The universal construction separates the workload from operating system threads and allows a dynamic assignment. We use helping to compensate for the failure of threads by reassigning the workload and to achieve maximal non-blocking progress.

2.1 Introduction

Multicore processors are commonplace and it is predicted that they will show asymmetric speeds of cores and a degraded reliability (see Section 1.1). Applications must rely on multi-threading for performance and effective abstractions are required to manage the complexity of concurrent and fault tolerant programming.

Most existing applications assign the parts of a given workload that can be parallelized to threads. Multiple threads traditionally synchronize using locks. However, locking-based techniques are subject to fault-tolerance problems, e.g., a thread may crash due to a hardware fault while holding locks, which blocks other threads waiting for that lock infinitely. A static assignment of the workload onto threads can lead to imbalances when the cores execute at asymmetric speeds. If a thread crashes, its share of the workload will not be processed and the result of the computation will be incomplete.

Wait-free implementations [Her91] of linearizable objects [HW90] circumvent the robustness issues of locking by ensuring a strong safety property (linearizability) together with a strong liveness property (wait-freedom). In short, linearizability provides the illusion of instantaneous access, whereas, wait-freedom ensures progress of every operation despite high contention and failures of other operations (see Section 1.2.2). Wait-free algorithms are, however, notoriously difficult to design and cannot be easily composed.

Transactional memory (TM) aims to simplify the synchronization of parallel applications compared to locks. A transaction is an operation with an explicitly delimited sequence of steps to be executed atomically by a single thread. A transaction can either commit (take effect) or abort (have no effect). With their fundamentally optimistic algorithm, progress of individual transactions becomes a crucial concern. There exist several types of TM designs [HLR10] in software, which differ in their liveness properties (e.g., lock-free, obstruction-free, blocking, see Section 1.2.3).

However, despite the many efforts deployed to ensure liveness of transactional memory systems using adequate contention management strategies and implementation designs, it has been shown recently [BGK12] that TMs cannot ensure a stronger liveness property than lock-freedom considering an asynchronous system with transaction crashes or non-terminating transactions, i.e., transactions that perform infinitely many operations without attempting to commit. In other words, one cannot implement a TM in the considered system model. Applying existing wait-free implementations [Her91] to resolve contention among conflicting transactions cannot guarantee wait-free progress when non-terminating transactions are considered.

The question we address in this chapter is whether it is possible to design a universal construction that is based on TM and guarantees wait-free progress for all correct operations while tolerating crashes and non-terminating operations. Therefore, we have to investigate if the impossibility result of [BGK12] is generalizable to all asynchronous systems, i.e., systems in which the absolute and relative speed of threads is not bounded and in which threads can
We describe an asynchronous system model, which we call AMSM (asynchronous multicore system model) that reflects the properties of standard, off-the-shelf multicore systems. The main restriction of AMSM is that we assume that the size of the memory is bounded. This is based on the fact that all processors have a fixed number of address lines to access the memory. We explain that one can implement a wait-free TM that tolerates thread crashes and non-terminating transactions. The main difference between AMSM and the model of [BGK12] is that AMSM permits transactions to be executed by any thread.

The rest of the chapter is organized as follows. We discuss the related work in Section 2.2. Section 2.3 introduces our system model (AMSM) and several definitions. Section 2.4 shows with help of the universal transactional memory construction that one can guarantee wait-free progress in the AMSM with TM. We give an informal proof in Section 2.5 and conclude the chapter in Section 2.6.

2.2 RELATED WORK

The universal construction was originally proposed by Herlihy [Her91; HS08]. It provides a wait-free implementation of a single concurrent object. Consensus is solved by appending invocations to an announce array that defines a log of invocations. All threads work on a private copy of the object and have to apply all invocations of the log. Thus, a thread has to perform the work of all other threads until it can apply its own invocation. Several proposals targeted the parallel execution of concurrent operations (e.g., [Bar93; TSP92]) but achieved only lock-free progress. The proposal by Moir [Moi97] is wait-free but requires a multi-word compare-and-swap implementation for its helper mechanism introducing an extensive locking scheme and prohibiting read-only concurrency. None of the proposed universal constructions take non-terminating invocations into account. Since the log is processed in order, an invocation that does not terminate would prevent subsequent invocations that were appended to the log from being executed.

Many non-blocking implementations direct accesses to shared objects to private copies of the object [HS08]. An example from the field of TM is DSTM [Her+03], an object-based and obstruction-free implementation. The benefit is that no intermediate data becomes visible until the operation completes with the cost of expensive copying for larger objects. Applying the local changes wait-free to the global object results in the consensus problem similar to the universal construction. Word-based TM implementations (e.g. [DSS06; Fel+10; MM08a]) reduce the copying overhead by keeping only modified memory locations in a log. The memory is protected by an array of locks, each protecting a number of memory locations. When two or more transactions contend for the same lock, one of them might need to be aborted. The notion of a contention manager [Gue+05; Her+03; SS04] has precisely been introduced as a modular mechanism to determine which transactions should wait or be aborted upon conflict. To achieve wait-free progress, no transaction is allowed to be continuously aborted. ROBUSTM [Wam+10] (see Chapter 3) guarantees that all transactions that are neither non-terminating nor crashed will terminate within a finite number of steps. This guarantee is achieved by eventually prioritizing transactions such that they win all conflicts while tolerating other non-terminating transactions. Its inherent limitation towards thread crashes results from the multi-threaded programming model: Once a thread crashed, all work already assigned to that thread will not be finished because no context information is available to allow a decoupling of transactions from threads.

Bushkov et al. [BGK12] consider an asynchronous system model that takes crashes and non-terminating transactions into account. Thread crashes execute a finite number of steps but can halt in the middle of a transaction. Non-terminating transactions, called parasitic, execute an infinite number of steps but starve the thread because they never attempt to commit. They assume a TM implementation that ensures the safety property of opacity [GK08], which
returns always a consistent value upon state access. The strongest TM-liveness property is local progress, which corresponds to lock-free progress in terms of committed transactions. Their impossibility result shows that local progress cannot be guaranteed in the presence of crashed or parasitic transactions while ensuring opacity. Ellen et al. [Ell+12] address the progress for universal constructions that expose disjoint access parallelism, which is in the nature of most TM implementations. They come to the impossibility result that both disjoint access parallelism and wait-freedom are not possible when the operations do not have a bound on the number of accesses to different data items. We show in this chapter that relying on a slightly different set of assumptions but, we believe, yet very practical assumptions (see Section 2.3) allows us to build a universal construction that avoids these limitations and allows a wait-free TM implementation.

2.3 ASYNCHRONOUS MULTICORE SYSTEM MODEL

In this section, we introduce the asynchronous multicore system model (AMSM) that aims at capturing the properties of current multicore and multi-processor systems. The system components are illustrated in Figure 2.1.

2.3.1 THREADS AND SYNCHRONIZATION PRIMITIVES

Threads are the basic entities of tasks that are created, managed and destroyed by the operating system (OS). While an application is a collection of instructions, a thread is the execution of the application in sequential steps when it is scheduled by the OS for the consumption of CPU cycles. Multiple threads can concurrently execute the application and communicate using shared memory. The scheduling of the threads is managed entirely by the OS, which in turn guarantees that every thread eventually gets execution time. Accesses to shared memory are encapsulated in transactions. The concurrent execution of threads is asynchronous, i.e., no upper bound is given on the relative speed of any two steps performed by different threads. However, each thread has a performance counter that holds the number of steps the thread has executed.

With application code being executed by threads, we have to be able to cope with thread crashes to ensure liveness. A thread is crashed if it stopped taking steps. In an asynchronous system, it is impossible for a thread to differentiate between another thread being crashed or just slow, because their relative speed is unbounded. To ensure liveness for the application execution, we assume in our system model that at least one thread is non-crashed. If all threads are crashed, no progress can take place.

A thread crash might occur if, for example, a system administrator permanently suspends a thread. We do not assume that a thread crash failure is detectable. A program can have bugs that lead to a crash of a thread (e.g., when trying to dereference a null pointer). We assume that these thread crashes are detected by the runtime system and converted in run-time exceptions, which can be detected.

Since a wait-free transactional memory system allows solving the consensus problem, threads need to have access to a synchronization primitive powerful enough to solve consensus. Current CPUs provide various synchronization primitives. In AMSM, we assume that threads have access to a Compare-And-Swap (CAS) primitive: \texttt{bool CAS(addr, expect, new)}. CAS takes three arguments: (1) a pointer to a shared memory location \texttt{addr}, (2) an expected value \texttt{expect}, and (3) the new value \texttt{new}. CAS will atomically read the value stored at \texttt{addr}, test if the read value is equal to \texttt{expect} and if this is the case, replace the contents of address \texttt{addr} with value \texttt{new}. It will return \texttt{true} on success and \texttt{false} if the memory location did not match the expected value. CAS is wait-free, i.e., it always terminates in a finite number of steps.
Figure 2.1: Under the AMSM, accesses to shared memory are protected by transactional memory. The code is executed by threads that are provided by the operating system. Crashes can occur for threads and the within the code. The invocations are organized in logs that will be processed by the threads.

For simplicity, we assume access to a Fetch-And-Increment (FAI) primitive that atomically adds 1 to an counter and returns its previous value. It is sufficient for FAI to be lock-free, which allows to implement the primitive using a loop that executes a CAS until it succeeds. In the loop, the current value of the counter is read and used as the expected value \( \text{expect} \). The new value \( \text{new} \) is the incremented read value.

### 2.3.2 INVOCATIONS AND TRANSACTIONS

Applications that want to benefit from the computational power of multicore CPUs have to harness the parallelism of the CPU. In our model, we use a terminology closely related to Herlihy’s universal construction [HS08]. All operations of an application are split in units of work that are called invocations. A sequence of invocations forms a log. A single log defines a total order on the sequence of its invocations similar to a FIFO queue, i.e., their execution must be sequential. Our model explicitly allows multiple logs at a time with the invocations of each log being executed in parallel. Appending invocations to a log must be a local operation because a log is a sequential object. In contrast to the original universal construction [Her91], our universal TM construction finds consensus not at the time invocations are appended but during the execution of invocations. Note that logs correspond to classical threads in the multi-threaded programming model where the developer defines a sequence of instructions for each thread that becomes a log of invocations in our model.

We interpret the shared memory as the sequential object to which the invocations are applied to and that the universal TM construction should transform into a wait-free linearizable object. The AMSM uses transactional memory (TM) to synchronize concurrent accesses during the parallel execution of invocations from different logs to shared memory (see Section 2.4). Therefore, a transaction encapsulates each invocation and the TM implementation guarantees a wait-free execution of all transactions. TM implementations provide an interface to start and commit transactions and to read and update shared data. We give only a brief overview of a TM API, details about the semantics can be found in the literature (e.g., [DSS06; FFR08; Her+03]). Before an application can access data that is stored in a shared memory...
location, it has to start a new transaction by performing a step \texttt{txBegin}. Within a transaction, memory locations can be read using \texttt{txRead} and modified using \texttt{txWrite}. All update operations of active transactions are performed on a local copy, keeping the shared memory in a consistent state. A correct transaction will try to commit after a finite number of steps by taking a step \texttt{txCommit}. After the completion of the commit step, all changes made by the transaction will atomically become visible in shared memory.

For simplicity, we assume that invocations only contain transactional code and the number of logs is known. Our model could easily be extended to support non-transactional code for invocations that operate only on the data of a closure, or for periods of time with only a single log for a truly serial execution.

Logs—or more precisely their invocations with encapsulated transactions—are executed by threads. We assume that the OS provides a set of threads of size \( n \) to execute \( k \) logs with \( n > k \). This means that at most \( k \) concurrent transactions can be active at a time. As the \( n \) threads are managed by the OS, the user-level TM implementation has no influence on their scheduling. The TM implementation instead controls the mapping of invocations to threads. Multiple threads are permitted to process the same invocations concurrently with the purpose of helping each other.

A transaction transforms upon successful commit the current state, which is represented by the shared memory, by applying its steps and producing a new state. Therefore, a transaction \( t \) is represented as a function \( f_{tx} \) such that \( S_{\text{new}} = f_{tx}(S_{\text{old}}) \). Since the memory of a computer is bounded, we assume that the number of states is also bounded (but the number of states is not necessarily known). Having a bounded number of states, applications can be represented as a \textit{finite state machine} (FSM). In the FSM, transactions form the transitions between states. Because we have a maximum of \( k \) logs, in each state of the AMSM there is a maximum of \( k \) possible transitions.

We permit threads to crash. At the time a \textit{thread crashes}, it might be executing instructions of an invocation. The crash of the thread will then propagate a crash of the invocation and its transaction. Since state modifications during the processing of a transaction are performed on a local copy, the transaction provides failure isolation in case the underlying thread crashes. The global state can only advance from one consistent state to another consistent state and happens atomically using a \textit{CAS} such that only one thread succeeds. To enable the application to cope with thread crashes, the TM implementation has to make sure that another thread takes over the processing of an unfinished invocation.

An invocation could also crash because of a programming bug, propagating its transaction to crash. We assume that such a \textit{transaction crash} will always result in an exception that is caught by the TM implementation. Such transactions will not affect the global state and the further processing of the associated invocation is discarded.

A \textit{non-terminating transaction} is a transaction that executes an infinite number of steps without attempting to commit. This is caused by an incorrect application program. Note that there could be other reasons why a transaction never commits, e.g., because the TM implementation continuously aborts the transaction before it has a chance to call the commit function. Our goal is that a TM implementation can tolerate non-terminating transactions while guaranteeing that “correct” transactions will eventually commit. The exact guarantees are defined with the help of liveness properties.

We use slightly stronger definitions of the liveness properties than those of Bushkov et al. [BGK12]. TM liveness properties describe which transactions of an application’s execution must commit. Only \textit{correct} transactions are covered. Bushkov et al. define that a transaction is \textit{correct} if it is neither non-terminating nor crashed. We strengthen this definition in the sense that a thread crash does not result in a transaction crash - instead a TM implementation should execute the affected transaction on a different thread. If a transaction crashes because the application code of the transaction crashes, we assume that this transaction is discarded.
Hence, we define that a transaction is *correct* if it is not non-terminating and all correct transactions must terminate within a finite number of steps.

Note that a thread can crash at any point in time. In particular, a thread can crash while executing code of the TM implementation.

### 2.4 UNIVERSAL TRANSACTIONAL MEMORY CONSTRUCTION

Our goal is to show that the wait-free liveness property for invocations can be satisfied in the asynchronous multicore system model (AMSM). This requires to show that each correct transaction, which represents an invocation, commits in a finite number of steps, even in the presence of crashed threads and non-terminating transactions.

In this section, we are going to explain the underlying idea of the universal TM construction. We first show how correct invocations are processed in a well-behaved environment with no thread crashes. We then extend our construction towards tolerating thread crashes as well as crashed and non-terminating transactions.

#### 2.4.1 UNIVERSAL CONSTRUCTION FOR THE GOOD CASE

We start with an overview of applications under the AMSM. An application comprises *k* logs that correspond to threads in the multi-threaded programming model. Each log consists of an ordered list of *invocations* with a monotonically increasing sequence number. Initially, each log is filled with a single invocation that will later append its successor to the same log. In this way, a chain of invocations is created that will be processed in FIFO order. Each invocation encapsulates a *transaction* for the synchronization of its accesses to shared memory. A transaction is a sequence of *steps*, which are calls to the TM API. Using the local sequence properties of a log, it is possible to define a predetermined total order on all transactions of the application. The *commit time* $CT$ of the $i$-th invocation ($0 \leq i$) in log number $l$ ($0 \leq l < k$) is calculated as follows: $CT - 1 = i \ast k + l$.

The shared memory is divided into *chunks* using a hash function in order to expose the disjoint access parallelism of an application. Each chunk contains all corresponding addresses and values that hash to the key it is assigned. The current global state is represented by a single object $S_{CT}$ that holds a reference to chunks with the current version for each key (see Figure 2.2). In addition to the current state, we keep a history of previous states in a list. Each state can be identified by its commit time. The current state can be advanced by appending a new state to the list using a CAS operation. All states in the history list are immutable. We will later show how one can perform garbage collection to bound the space needed for the history. At application start time, the shared memory is copied into the corresponding chunks and references to the chunks are stored in the initial state object $S_0$.

The operating system initially provides *n* physical *threads* to process the invocations of the application. For our universal construction, we need to map the invocations to threads for processing. This is solved by using a monotonically increasing counter $CT$ (initialized with 1) in the following way: Each time a thread is seeking for work it performs a FAI on the counter to obtain the commit time slot it is supposed to process (MAIN in Algorithm 2.1). Based on the value of $CT$, it finds the log to take the invocation from using $(CT - 1) \mod k$ and the invocation to process by the sequence number $(CT - 1) \div k$ (see Figure 2.3 and PROCESSINVOCATION in Algorithm 2.1). This way, processing the invocations in the predetermined total order is maintained. After identifying the invocation, the thread starts processing the invocation's transaction following the TM algorithm.

The goal of the universal TM construction is to execute all correct invocations within a finite number of steps. We allow concurrent processing and mask invalid state of crashed or non-terminating invocations with the help of transactional memory. A transaction is represented as
2.4 Universal Transactional Memory Construction

Figure 2.2: State representation in the universal TM construction. Chunks with a gray box hold modifications while white boxes point to a chunk with an older commit time.

Figure 2.3: Assignment of invocations to threads for processing. Transaction $f_{tx}$ operates on a private state $s_{ct}$ that only becomes the global state $S_{CT}$ when $f_{tx}$ and all proceeding states $S_{<CT}$ finished.
Algorithm 2.1: Thread functions for the processing of invocations.

function MAIN
  moreWork ← true // Main function module
  while moreWork do
    moreWork ← PROCESSINVOCATION(FAI(CT)) // Continue with next invocation
  
function PROCESSINVOCATION(ct) // Process a given invocation
  invocation ← LOGS[(ct − 1)%K].invocation[(ct − 1)/K] // Find invocation
  if ¬ invocation then // Is proceeding invocation finished?
    WAIT(ct − K) // Help
  
  try invocation.fx(ct)
  catch (SkipException) break
  catch (TimeoutException) APPEND(ct, false) throw
  catch (Exception) COMMITPROCEEDING(ct)
  APPEND(ct, true)
  
foreach log in LOGS do
  if ¬ log.invocation[ct/K].empty then // More invocations to process?
    return true

return false

function APPEND(ct, new) // Add result to global state
  if new then // Append new state?
    next ← log[ct%K].invocation[ct/K].successor
  else
    next ← log[ct%K].invocation[ct/K]
  CAS(log[s.ct%K].invocation[s.ct/K + 1], NULL, next)

function WAIT(ct) // Help until proceeding state available
  while S[CURRENT].ct < ct do
    PROCESSINVOCATION(S[CURRENT].ct) // Process previous invocation
  
function TIMEOUT(s) // Add result to global state
  if S[CURRENT].ct ≥ s.ct then // Already processed by another thread?
    DELETE(s)
    throw(SkipException)
    threshold ← THRESHOLD[s.ct%K, ct/K]
    if STEPS[threadId] ≥ threshold then // Threshold exceeded?
      THRESHOLD[s.ct%K, ct/K + 1] ← threshold + 1
      STEPS[threadId] ← 0
      DELETE(s)
    throw(TimeoutException)
a function $S_{CT} = f_{tx}(S_{CT-1})$. We expose parallelism by executing $f_{tx}$ concurrently with other transactions while speculating on a possibly future $S_{CT-1}$. The intermediate state during the execution of an invocation is private to its encapsulating transaction and not visible to other transactions. This way, invocations always operate on a consistent state. Any invocation can only be finalized after its proceeding state is available. If the proceeding state $S_{CT-1}$ is not available, the new state $S_{CT}$ cannot be applied. Instead, the thread has to wait and performs a helper mechanism that we introduce in Section 2.4.2. Our TM algorithm includes the following steps (summarized in Algorithm 2.2):

**START** first obtains a clone of the current head of the state history as a private state and sets the commit time to the same value as the processing thread. Cloning a global state only creates references to the chunks of the current state, no chunks are copied. It is safe because the states in the history are immutable. Note that the cloned state does not need to be the direct ancestor of the transaction’s commit time. The commit time of the cloned state is therefore kept as the base version.

**WRITE** first identifies the chunk where the address to update is located. It then checks if that chunk still points to the original state that was cloned. In that case, the chunk itself is cloned to allow the update of its values in the context of the transaction’s state.

**READ** identifies the chunk where the address to read is located and checks if it must keep the commit time of the value returned in the read-set. This is the case if the transaction has not written to the chunk and the chunk still points to a state that is not the direct ancestor of the transaction. Opacity is ensured because the current states being read is immutable and always consistent.

**COMMIT** has to wait until the state with the proceeding commit time is available (**WAIT** in Algorithm 2.1). It then obtains a reference to that state to perform the validation of the transaction. For each entry in the read-set it checks that the commit time of the chunk did not increase. If the validation failed, the transaction must be aborted and will be retried by the processing thread until it succeeds. Note that aborts can only happen until the proceeding commit time has been reached in the global state. When the validation was successful, all references of the local state to chunks that were modified after the cloning must be updated to point to their latest commit time. The last step is to add the state that has been created locally to the commit time slot in global history of states. The operations is performed using a CAS that expects the slot to be empty because only a single thread must succeed. This becomes significant when we later discuss our helper mechanism in Section 2.4.2.

**COMMITPROCEEDING** waits for its ancestor state, clones it and adds the state, which only consists of references to states with an earlier commit time, to the state history using CAS. The function is used whenever a transaction crashed or exceeded its timeout (see Section 2.4.3).

A thread that finished the processing of an invocation tries to append a successor invocation to the same log (**APPEND** and **PROCESSINVOCATION** in Algorithm 2.2). **APPEND** will either try to append the successor of the last complete invocation of the same log or it tries to append the same invocation in case its processing was not completed. Since an invocation can only be appended to the same log after its predecessor has been processed, **PROCESSINVOCATION** might have to wait if it is assigned a commit time that targets an invocation that is not available yet.

The application will not terminate as long as at least one log contains further invocations (**MAIN** and **PROCESSINVOCATION** in Algorithm 2.1). If the log $CT \mod k$ is finished it will append
Algorithm 2.2: Transactional memory algorithm.

1. function \texttt{START}(ct, readSet) \hfill \texttt{// Start a new transaction}
   2. \texttt{CLEAR}(readSet)
   3. \texttt{s} $\leftarrow$ \texttt{CLONE}($S[\texttt{CURRENT}]$)
   4. \texttt{s.base} $\leftarrow$ \texttt{s.ct}
   5. \texttt{s.ct} $\leftarrow$ \texttt{ct}
   6. return \texttt{s}

7. function \texttt{WRITE}(s, addr, val) \hfill \texttt{// Update the private state}
   8. \texttt{TIMEOUT}(s)
   9. \texttt{key} $\leftarrow$ \texttt{hash}(addr)
   10. \texttt{chunk} $\leftarrow$ \texttt{s.chunk}[key]
   11. if \texttt{chunk.ct} < \texttt{s.ct} \texttt{// Chunk not cloned yet?}
   12.     \texttt{s.chunk}[key] $\leftarrow$ \texttt{CLONE(chunk)}
   13.     \texttt{chunk} $\leftarrow$ \texttt{s.chunk}[key]
   14.     \texttt{chunk.ct} $\leftarrow$ \texttt{s.ct}
   15.     \texttt{chunk.data}[addr] $\leftarrow$ \texttt{val}

16. function \texttt{READ}(s, readSet, addr) \hfill \texttt{// Read from the global state}
   17. \texttt{TIMEOUT}(s)
   18. \texttt{key} $\leftarrow$ \texttt{hash}(addr)
   19. \texttt{chunk} $\leftarrow$ \texttt{s.chunk}[key]
   20. if \texttt{chunk.ct} < \texttt{s.ct} and \texttt{s.base} $\neq$ \texttt{s.ct} - 1 \texttt{// Speculative read?}
   21.     \texttt{readSet[key].base} $\leftarrow$ \texttt{chunk.ct}
   22. return \texttt{chunk.data}[addr]

23. function \texttt{COMMIT}(s, readSet) \hfill \texttt{// Commit a transaction}
   24. \texttt{TIMEOUT}(s)
   25. \texttt{WAIT}(s.ct)
   26. \texttt{p} $\leftarrow$ \texttt{S[s.ct - 1]}
   27. foreach each \texttt{key} in \texttt{readSet} do
   28.     if \texttt{p.chunk}[key].ct > \texttt{readSet[key].base} \texttt{// Outdated chunk?}
   29.         \texttt{jump(START(s.ct, readSet))}
   30.     if \texttt{~CAS}(S[s.ct], \texttt{NULL}, s) \texttt{// Try to append state}
   31.         \texttt{DELETETE(s)}
   32. \texttt{CAS}(\texttt{CURRENT}, \texttt{s.base}, \texttt{s.ct})

33. function \texttt{COMMITPROCEEDING}(ct) \hfill \texttt{// Commit a proceeding state}
   34. \texttt{WAIT}(ct)
   35. \texttt{s} $\leftarrow$ \texttt{CLONE}(S[ct - 1])
   36. \texttt{s.base} $\leftarrow$ \texttt{s.ct}
   37. \texttt{s.ct} $\leftarrow$ \texttt{ct}
   38. if \texttt{~CAS}(S[ct], \texttt{NULL}, s) \texttt{// Try to append state}
   39.     \texttt{DELETETE(s)}
   40. \texttt{CAS}(\texttt{CURRENT}, \texttt{s.base}, \texttt{s.ct})
2.4 Universal Transactional Memory Construction

![Diagram of transactional memory with thread and transaction crashes](image)

Figure 2.4: Dealing with a) thread and b) transaction crashes. Thread crashes are not detectable under the AMSM and require helping from threads that are assigned succeeding commit times. Transaction crashes throw an exception and are considered as persistent failures. The proceeding state is committed and no further invocations can be appended to the log.

only empty invocations. If an invocation at $CT \div k$ is empty the thread will execute a transaction that simply commits the predecessor state $S_{CT-1}$.

Each successfully processed invocation will add a new object to the history of states. The space used by the history should be bound in order to preserve memory. Therefore, garbage collection is performed periodically to remove obsolete states, chunks and invocations. The period could be based on transaction commits. The operating system provides $n$ threads, so maximal $n$ different invocations can be processed at a time. Since invocations are executed in order based on $CT$, the oldest proceeding state we need to keep is $S_{CT-k}$. When discarding an older state during garbage collection, references to chunks with an older commit time than the state are removed (white boxes in Figure 2.2). Chunks include a reference counter and if it reaches zero while a reference is removed, the chunk can be deleted. Garbage collection for processed invocations can be performed in a similar way.

2.4.2 DEALING WITH THREAD AND TRANSACTION CRASHES

The AMSM allows threads to crash at any time but a crash cannot be detected. The TM implementation must make sure that all invocations that were processed by a crashed thread are assigned to another thread, as illustrated in Figure 2.4 a). All threads that wait in COMMIT for the proceeding state $S_{CT-1}$ to become available help processing all pending invocations. Therefore, the threads identify the oldest pending invocation by the commit time of the current global state (see WAIT in Algorithm 2.1). One thread will successfully finish processing of an invocation and commit its transaction to the global state using a CAS while the other threads will discard their changes whenever they fail the CAS. It is sufficient that the apply operation for $S_{CT}$ is lock-free because all threads try to append equivalent states for a specific $CT$.

With the helper mechanism, thread crashes are naturally masked by non-crashed threads and all invocations will be processed as long as at least one thread survives and runs long enough to commit a transaction. For optimization reasons, threads that execute the helper mechanism should periodically check whether another thread already finished processing a pending invocation (see TIMEOUT in Algorithm 2.1). In that case, it stops processing the current invocation by throwing a SkipException.
In contrast to thread crashes, crashes of transactions can be detected by an exception. Transaction crashes are considered as persistent failures and must be caught because otherwise they would propagate to a thread crash. This would result in all other threads to crash because of our helper mechanism. Instead, the thread intercepts the Exception during PROCESS_INVOCATION and commits the proceeding state such that $S_{CT} = S_{CT-1}$, as shown in Figure 2.4 b). We assume that succeeding invocations that are appended to the same log perform an empty transaction.

2.4.3 TOLERATING NON-TERMINATING TRANSACTIONS

Transactions that perform infinitely many steps without ever trying to commit are called non-terminating transactions. Problems for wait-free progress arise when a thread tries to execute the function $f_{tx}$ of a non-terminating transaction: The function will never return. Since all threads execute all logs with the helper mechanism, all threads would be executing forever the first non-terminating function encountered. To circumvent this problem, we assign each invocation a quota of steps which it is allowed to execute. If it exceeds its quota, the processing is stopped and retried with a higher quota after invocations of other logs had a chance to complete. This mechanism effectively bounds the length of transactions, which is essential to achieve a wait-free progress guarantee for disjoint access parallelism in the form of dynamically overlapping transactions [Ell+12].

In detail, we define a maximum number of steps (stored in a variable threshold[$l$, $i$], see TIMEOUT in Algorithm 2.1) that a thread will execute of the $i$th invocation of log $l$ before starting to execute the invocation of the next log $l + 1$ mod $k$. Each thread can read the number of steps it executed from a performance counter. Invocations that complete within the given number of steps proved to be correct and their resulting state got appended to the global state history. If the invocation did not terminate, its transaction is aborted by a TimeoutException and the proceeding state is applied to the state history. While the mechanism so far equals the handling of crashed transactions, we now need to re-append the same invocation to the same log after incrementing threshold[$l$, $i$ + 1]. The threshold will eventually increase to be large enough to commit every correct transaction. Note that calls to TIMEOUT must be added periodically (in particular in loops, i.e., by a compiler) if invocations contain non-transactional code or insufficient calls to the TM runtime.

More formally, for each correct transaction there exists a constant $C$ that a transaction will take at most $C$ steps to terminate. $C$ is unknown and usually quite large. We try to estimate $C$ by increasing strictly monotonically the number of steps a transaction can take such that for correct transactions the number of steps will be sufficiently large. Since we have a bounded number of states, a correct transaction has to terminate within a bounded number of steps. Otherwise, during the processing of a invocation’s transaction, the execution would traverse the same state at least twice. If that happens, we have a cycle in the execution, i.e., this would be a non-terminating transaction. Because we increase the number of permitted steps every time we reach the threshold for a correct transaction, eventually the variable threshold[$l$, $i$] will become sufficiently large to execute a correct transactions of invocations $i$ of log $l$.

2.5 PROOF OF CORRECTNESS

In this section, we will informally proof the correctness of our assumptions for the universal TM construction.

Theorem 1. The length of the state history is bound by the number of threads $n$.

Proof. Threads can only obtain a new commit time slot $ct$ after $S_{ct-n}$ has been added to the history. Threads that are very slow and did not discover the state update and try to
access a state older than $S_{ct-n}$ in the transaction will be notified by an exception. During exception handling the thread calls COMMITPROCEEDING but both CAS operations will fail because another thread already succeeded.

**Theorem 2.** The universal TM construction processes $k$ invocations in parallel, where $k$ is the number of logs and at least $k$ threads are not crashed.

**Proof.** Each thread is mapped to a log based on $ct \mod k$. Since every thread obtains a monotonically increasing commit time using $ct = FAI(CT)$ no two threads will have the same $ct$. For each thread $ct$ will process an invocation in the range from $f_{tx}(CT-k-1)$ to $f_{tx}(CT-1)$. Thus, each of $k$ threads will process a different log out of $k$ logs. If the total number of threads $n > k$, more than one thread will be mapped to the same log and each will process the same invocation of that log because no successor in the range from $f_{tx}(CT)$ to $f_{tx}(CT + n - k - 1)$ is available yet. Threads that perform the helper mechanism regularly check if the state they are supposed to help computing already got available and return in this case to their original $ct$.

The level of parallelism that can be achieved depends on the disjoint access parallelism available in the application and in the difference of the number of steps of invocations. The larger the difference is, the higher the time threads will perform the helper mechanism in order to maintain the sequential order of commits. The optimization of the chosen hash function and the number of keys is analogue to the optimization of modern word-based TMs that use a similar partitioning of shared memory for internal synchronization.

**Theorem 3.** All correct transactions complete in finite steps if no thread crashes are present.

**Proof.** A non-crashed thread will retry a transaction until it commits. Transactions are totally ordered by $CT$, thus, a transaction $f_{tx}(S_{CT-1})$ can only abort as long as $S_{CT-1}$ is not available and it has to speculate on a previous state. $S_0$ is present as the initial state of the application and $f_{tx}(S_1)$ cannot abort because $S_0$ is like all other states in the history immutable. Since all correct transaction reach COMMIT in a finite number of steps after their immutable predecessor state became available, $f_{tx}(S_{CT-1})$ reaches COMMIT in a finite number of steps after $S_{CT-1}$ got appended eventually to the history.

**Theorem 4.** The universal wait-free progress guarantee tolerates $n-1$ thread crashes.

**Proof.** Our construction includes a helper mechanism that is enabled when $f_{tx}(S_{CT-1})$ waits for $S_{CT-1}$ in COMMIT to perform its validation. All threads that wait will try to compute all states between $S_{current}$ and $S_{CT-1}$. As long as one thread survives and runs long enough to complete $f_{tx}(S_{CT-1})$, crashes of all other threads can be tolerated.

**Theorem 5.** Non-terminating and crashed transactions do not stop correct transactions from making progress.

**Proof.** Crashed transactions can be directly detected by an exception and their encapsulating invocation is discarded from further processing. Non-terminating transactions will only be processed for threshold$[l, i]$ steps. In both cases, $S_{CT} = S_{CT-1}$ becomes available by COMMITPROCEEDING instead of $S_{CT} = f_{tx}(S_{CT-1})$ in a finite number of steps allowing all other logs that depend on $S_{CT}$ as a proceeding state to continue making progress.

While the universal TM construction tolerates crashes and non-terminating transactions, their presence can delay the execution of correct transactions. ROBUSTM [Wam+10] (see Chapter 3) follows a similar approach to tolerate non-terminating transactions and experiments showed that after a warm-up phase a good estimate for the threshold can be found, such that it is large enough to cover the number of steps of typical transactions for the application’s workload.
Theorem 6. The universal TM construction terminates all correct invocations in finite steps.

Proof. Follows from Theorems 3, 4, and 5, because the universal construction is based on the wait-free TM and all invocations are embedded in transactions.

2.6 SUMMARY

We have shown that the universal TM construction is capable of processing all correct invocations of multiple logs under the assumptions of the AMSM while preserving the wait-free progress guarantee. The liveness property holds in the presence of thread crashes and crashed or non-terminating invocations. These symptoms can be the effect of hardware faults.

The response of the invocations is applied to the global state object that is protected by transactional memory in the order of their predetermined commit time, eliminating the need for a contention manager. All read-only operations can be performed in parallel. The amount of copied data for write operations is significantly reduced in comparison to the original universal construction. Dividing the shared memory with a hash function as known from word-based TM implementations allows to expose disjoint access parallelism. In contrast to word-based TM implementations that rely on ownership records or locks, our construction is truly non-blocking based on a single CAS to apply a completed operation to the global state.

The universal TM construction could be greatly simplified with AMD’s Advanced Synchronization Facility (ASF) [Chr+10] - however, ASF is unfortunately not available in silicon. ASF could be used to monitor the state history either to stop the helper mechanism as soon as the state was added to the history by another thread or to abort a transaction as soon as a chunk it read was updated by a proceeding transaction. An alternative approach would be to use ASF to shrink the state history. So far, a transaction first clones the current state by creating references to it in order to keep its modifications local until it atomically tries to add the new state to the history using a single CAS. ASF would allow to swap multiple chunks atomically such that instead of the entire state only individual chunks are cloned. Chunks that were not cloned during an update operation always point to the current state. No history of states would be needed.
3 ROBUSTM: A ROBUST SOFTWARE TRANSACTIONAL MEMORY*

*The contents of this chapter first appeared at SSS ’10 [Warn+10].
In this chapter, we shift the focus from the fault-tolerant universal construction that executes all code in transactions towards using transactions solely for a robust synchronization.

For software transactional memory to be usable in large applications such as databases, it needs to be robust, i.e., live, efficient, tolerant of crashed and non-terminating transactions, and practical. In this chapter, we study the question of whether one can implement a robust software transactional memory in an asynchronous system. To that end, we introduce a system model — the multicore system model (MSM) — which captures the properties provided by mainstream multicore systems. We show how to implement a robust software transactional memory (ROBUSTM) that provides maximal blocking progress in MSM. Our experimental evaluation indicates that ROBUSTM compares well against existing blocking and non-blocking software transactional memories in terms of performance while providing a much higher degree of robustness.

3.1 INTRODUCTION

As a consequence of the establishment of software transactional memory (STM) as a viable synchronization mechanism, our general goal is to investigate the use of STM in large software systems like application servers, databases, or operating systems. Such systems are developed and maintained by hundreds of programmers, and all that code lives in the same address space of the system’s process. Ensuring the robustness of such applications requires the use of techniques that guarantee the recovery from situations in which individual threads crash or behave improperly (e.g., loop infinitely) while executing critical sections. For example, commercial databases guarantee such robustness using custom mechanisms for lock-based critical sections [Lah+01].

A system that uses transactions to perform certain tasks typically relies on their completion. Thus, a robust STM must guarantee that all well-behaved transactions will terminate within a finite number of steps. A transaction is well-behaved if it is neither crashed nor non-terminating (see Section 2.3.2). Both crashed and non-terminating transactions can interfere with the internal synchronization mechanism of the underlying STM implementation, possibly preventing other transactions from making progress if not handled correctly. A crashed transaction will stop executing prematurely, i.e., it executes a finite number of steps and stops before committing (e.g., due to failure of the associated thread). A non-terminating transaction executes an infinite number of steps without attempting to commit.

Note that a robust STM provides guarantees that are very similar to a wait-free STM, which guarantees to commit all concurrent transactions in a bound number of steps. Yet, the definition of the wait-free property requires the use of an asynchronous model of computation, but it has been shown recently [BGK12; Ell+12] that one cannot implement a wait-free STM in such a system model. However, current multicore systems provide stronger guarantees than those postulated in the asynchronous system model. Therefore, we try to answer the question whether one can implement a robust STM in today’s multicore computer architectures.

We showed in Chapter 2 how to build a universal construction that tolerates crashes and non-terminating transactions, e.g., as symptoms from hardware faults. Unfortunately, the guarantee for wait-free progress does neither state nor promise any practical throughput level. In contrast to a universal construction, a robust STM focuses solely on the synchronization that must also be efficient and makes no assumptions about the programming model.

In this chapter, we introduce a new multicore system model (MSM). It is asynchronous in the sense that it does not guarantee any bounds on the absolute or relative speed of threads but additionally reflects the properties of mainstream multicore systems. We show that one can implement a robust STM (ROBUSTM) in MSM that guarantees progress for individual threads. Our ROBUSTM implementation exhibits performance comparable to state-of-the-art lock-based STMs on various types of benchmarks. Therefore, we not only show that one can
implement robust STMs but also that one can implement them efficiently.

The chapter is organized as follows: We first discuss related work in Section 3.2. We introduce MSM in Section 3.3. Section 3.4 presents the algorithm of ROBUSTM. We evaluate our approach in Section 3.5 and conclude in Section 3.6.

3.2 RELATED WORK

Non-blocking concurrent algorithms (e.g., lock-free or wait-free) ensure progress of some or all remaining threads even if one thread stops making progress. While many early STMs where non-blocking, most of the recent implementations use blocking algorithms because of their simpler design and better performance. Recent work on non-blocking STM [MM08b; Tab+09] has shown that its performance can be substantially increased by applying techniques known from blocking STM implementations. This includes (1) timestamp-based conflict detection and (2) a reduced number of indirections while operating on transactional data by accessing memory in place in the common case. Depending on the algorithm, costly indirection is still required either during commit [MM08b] or when stealing ownership records on conflict [Tab+09]. For the later, deflating the indirection is only possible after the original owner transaction moved to the abort state, but this might never happen for transactions that are not well-behaved.

Contention management was originally introduced to increase the throughput and avoiding possible livelocks (e.g., Polite, Karma, Polka [SS04; SS05]). An interesting observation is to back off the losing transaction after a conflict to avoid encountering the same conflict immediately upon retry. Contention managers that aim to provide fairness between short and long-running transactions usually rely on prioritization. The priority can be derived from the time when a transaction started or the amount of work it has done so far [GHP05; SS05]. This helps long-running transactions reach their commit point but can delay short transactions extensively in case of high contention. Furthermore, crashed transactions will gain a high priority if it is based on the start time. An alternative is to derive the priority from the number of times the transaction has already been retried [SS04] and favor transactions with problems reaching their commit point.

In combination with priorities, simple mechanisms such as recency timestamps or liveness flags were introduced to determine the amount of time that contending transactions should back off. The goal is to increase the likelihood that a transaction that has already modified a memory location can commit (e.g., Timestamp, Published Timestamp [SS04; SS05]). These mechanisms are also used by transactions to show that they are not crashed. However, this approach does not work for non-terminating transactions because they may well update the timestamp or flag forever. The length of potential contention intervals can be reduced if locks are not acquired before commit time [Spe+09a]. This would allow us to tolerate non-terminating transactions because they never try to commit [BGK12], but by detecting conflicts lazily one cannot ensure that a transaction will eventually manage to commit (it can be repeatedly forced to abort by concurrent transactions that commit updates to shared memory).

Contention managers can also try to ensure progress of individual transactions. In the initial proposal of the Greedy contention manager [GHP05], which guarantees that every well-behaved transaction commits within a bounded amount of time, thread failures could prevent global progress (i.e., the property that at least some thread makes progress). This issue was solved by giving each transaction a bounded period of time during which it could not be aborted by other transactions [Gue+05]. If a correct transaction exceeds this time limit and is aborted, it can retry with a longer delay. This approach works for crash failures but not for non-terminating transactions because the delay can grow arbitrarily large if the transaction is retried infinitely often.
Fich et al. [Fic+05] proposed a transformation algorithm that converts any obstruction-free algorithm [Her+03] into a practically wait-free one. The idea is that, in a semi-synchronous system, it is impossible to determine if a thread has crashed by observing its executed steps, as a step can take a bounded but unknown amount of time to complete. Thus, it is not possible to know a priori how long to wait for a possibly crashed transaction. Instead, one has to wait for increasingly longer periods. To decide if a thread had indeed crashed after expiration of the waiting period, they observe the instruction counter of the thread used to track progress. This approach cannot be applied straightforwardly to STMs because transactions can contain loops or perform operations with variable durations, e.g., allocate memory, so we cannot automatically and efficiently determine the abstract linear instruction counter of a running transaction.

Bushkov et al. [BGK12] explicitly take non-terminating transactions into account, called parasitic processes. Their result is that the strongest progress guarantee that can be ensured in asynchronous systems is global progress, which is analogous to lock-freedom. Since thread crashes and non-terminating transactions are not detected but tolerated, one cannot give to a single transaction an exclusive execution right because the thread might gain the right and never release it. We show in this chapter that relying on a different but yet practical system model (see Section 3.3) allows us to build robust STMs that avoid these limitations and work on current multicore systems.

Ellen et al. [Ell+12] proved that no universal construction that ensures disjoint access parallelism can be wait-free, except if the operations have a bounded length that restricts the different accessed data items. Attiya et al. [AHM09] showed that invisible reads prevent wait-free progress for disjoint access parallel STM algorithms. Our design exposes transactions in different modes for efficiency that make reads only visible progress is at stake and limit the length of transactions when they run privileged.

### 3.3 SYSTEM MODEL

Our multicore system model (MSM) satisfies the following nine properties. (1) A process consists of a non-empty set of threads that share an address space. (2) All non-crashed threads execute their program code with a non-zero speed. Neither the absolute nor the relative speed of threads is bounded. (3) Threads can fail by crashing. A crash can be caused by a programming bug or by a hardware issue. In the case of a hardware issue, we assume that the process crashes. In case of a software bug, only a subset of the threads of a process might crash. (4) We assume that STM is correctly implemented, i.e., crashes of threads are caused by application bugs and not by the STM itself. The motivation is that a STM has typically a much smaller code size that is reused amongst multiple applications. (5) A process can detect the crash of one of its threads. (6) Threads can synchronize using CAS and atomic-or operations (see below). (7) The state of a process is finite. (8) A thread can clone the address space of the process. (9) Each thread has a performance counter that counts the number of instructions it executed.

Compared to the asynchronous multicore system model from Section 2.3, the MSM reflects only the properties of modern hardware and operating systems and makes no assumptions about the application structure. While the practical assumptions include the thread crash detection and cloning, the decoupling of the workload from threads is left to the application to ensure that the computation can be completed if threads fail. Possible application-level solutions include dynamic workload rebalancing or the dispatching of tasks into queues for a processing by a thread pool.

**Software Transactional Memory.** In our model, we assume that transactions are executed concurrently by threads. Within transactions, all accesses to shared state must be redirected to the STM and neither non-transactional accesses to global data nor external IO operations
are permitted. If a thread failed to commit the transaction, it is retried (see Section 3.4). We further assume that transactions are non-deterministic and allow transactions to execute different code paths or access different memory locations during the retry.

Detection mechanisms. Modern operating systems permit the detection of thread crash failures. A thread can crash for various reasons like an uncaught exception. To detect the crash of a thread that is mapped to an operating system process, one can read the list of all processes that currently exist and check their status or search for missing threads [AW09]. The MSM assumes the existence of a thread crash detector that detects within a finite number of steps when a thread has crashed (i.e., the thread stopped executing steps) and will not wrongly suspect a correct thread to have crashed. For simplicity, in our implementation we assume that a signal handler is executed whenever a thread crashes.

Progress mechanisms. Like many concurrent algorithms, the MSM assumes the existence of a compare-and-swap operation (CAS, see Section 2.3.1). A CAS is often used in loops in which a thread retries until its CAS succeeds (see Algorithm 3.1). Note that sometimes such a loop might contain a contention manager to resolve a conflict with another thread but in the meantime a third thread might have successfully changed $\text{addr}$. In other words, a contention manager might not be able to ensure progress of an individual thread since this thread might have continuous contention with two or more other threads.

**Algorithm 3.1:** While CAS is wait-free, there is no guarantee that the CAS will ever succeed, i.e., that the loop ever terminates.

1. repeat
2. \[ \text{expect} \leftarrow \ast \text{addr} \] // Read current value
3. \[ \text{new} \leftarrow \text{FUNCTION(expect)} \] // Get new value
4. until $\text{CAS}($addr, expect, new$)$

**Algorithm 3.2:** Using an atomic-or, we can make sure the CAS of the privileged priority thread always succeeds.

1. repeat
2. \[ \text{if has_priority then} \] // Privileged priority
3. \[ \text{atomic-or}($\text{addr}, F) \] // Set fail bit
4. \[ \text{expect} \leftarrow \ast \text{addr} \] // Expect bit in CAS
5. \[ \text{else} \] // All other threads
6. \[ \text{expect} \leftarrow \ast \text{addr} \& \sim F \] // No fail bit
7. until $\text{CAS}($addr, expect, new$)$

The problem is that there is no guarantee that a thread will ever be successful in performing a CAS. To address this issue, the MSM assumes an atomic-or operation. Note that the x86 architecture supports such an operation: a programmer can just add a \texttt{LOCK} prefix to a logical or operation. It is guaranteed, that a processor will execute the atomic-or operation in a finite number of steps. Also note that such an operation does not exist on, for example, Sparc processors.

We use the atomic-or to ensure that each correct transaction will eventually commit. ROBUSTM will select at most one thread with a privileged priority level in the sense that this thread should win all conflicts. To ensure that all CAS operations performed by a privileged thread succeed, it uses the atomic-or to make sure that all competing CASEs fail. To do so, we reserve a bit ($F$) in each word that is used with a CAS (see Algorithm 3.2). If a privileged thread performs an atomic-or just before another thread tries to perform a CAS, the latter will fail because its expected value assumes the $F$ bit to be cleared.
Our goal is not only to implement wait-free transactions in the face of crash failures, but also in the face of non-terminating transactions. We assume however that the STM code itself is well-behaved and only application code can crash or loop infinitely often. For tolerating non-terminating transactions, we assume two more mechanisms that can be found in current systems. First, the MSM assumes that we can clone a thread, i.e., the operating system copies the address space of a process (using copy-on-write) and the cloned thread executes in a new address space fully isolated from all threads of the original process. Second, the MSM assumes the existence of a performance counter that (1) counts the cycles executed by a thread, and (2) permits other threads to read this performance counter. The intuition of the performance counter is as follows. The privileged thread can keep its privilege for a certain number of cycles (measured by the performance counter), after which it is not permitted anymore to steal the locks of other threads. If we can prove that the thread is well-behaved and would have simply needed more time to terminate, we increase the time quantum given to the privileged thread. Since the state space of threads is finite (but potentially very large), there exists a finite threshold $S$ such that each transaction will either try to commit in at most $S$ steps, or it will never try to commit. The problem is how to determine an upper bound on this threshold for non-deterministic transactions (see Section 3.4). Our system ensures that non-terminating transactions are eventually isolated to ensure the other threads can make progress while ensuring that long running but correct transactions will eventually commit.

3.4 DESIGN AND IMPLEMENTATION

Our STM algorithm runs in different modes. In this section, we first present the basic algorithm optimized for the good case with well-behaved transactions (Mode 1). When conflicts are detected and fairness is at stake, we switch to Mode 2 by prioritizing transactions. If the system detects a lack of progress, we switch to Mode 3 for dealing with crashed and non-terminating transactions. The mode is set for each transaction individually.

3.4.1 WHY A LOCK-BASED DESIGN?

Our robust STM algorithm uses a lock-based design. The reason for basing our work on a blocking approach instead of an obstruction-free one is driven by performance considerations. Non-blocking implementations suffer from costly indirections necessary for meeting their obstruction-free progress guarantee [DSS06; Enn06; MM08b]. Although many techniques known from blocking implementations were applied to avoid indirection under normal operation with little contention, indirection is still necessary when it comes to conflicts with transactions that are not well-behaved (see Section 3.2). Our own experiments (see Section 3.5) still show a superior performance of lock-based designs.

Several reasons can explain the good performance of blocking STMs. They have a simpler fast path and more streamlined implementations of the read/write operations with no extra indirection. In addition, the combination of invisible reads and time-based validation [RFF06] provides significant performance benefits. In this chapter, we use a C++ version of the publicly-available TINYSTM [FFR08] as the basis for our robust STM algorithm. TINYSTM is an efficient lock-based implementation of the lazy snapshot algorithm (LSA) [RFF06].

3.4.2 OPTIMIZING FOR THE GOOD CASE

For completeness, we briefly recall here the basic algorithm used by TINYSTM. Like several other word-based STM designs, TINYSTM relies upon a shared array of locks to protect memory from concurrent accesses (see Figure 3.1). Each lock covers a portion of the address space. In our implementation, it uses a per-stripe mapping where addresses are mapped to locks based on a hash function.
Each lock is the size of an address on the target architecture. Its least significant bit is used to indicate whether the lock has been acquired by some transaction. If it is free, the STM stores in the remaining bits a version number that corresponds to the commit timestamp of the transaction that last wrote to one of the memory locations covered by the lock. If the lock is taken, the STM stores in the remaining bits a pointer to an entry in the write-set of the owner transaction. Note that addresses point to structures that are word-aligned and their least significant bits are always zero on 64-bit architectures; hence one of these bits can safely be used as lock bit.

When writing to a memory location, a transaction first identifies the lock entry that covers the memory address and atomically reads its value. If the lock bit is set, the transaction checks if it owns the lock using the address stored in the remaining bits of the entry. In that case, it simply writes the new value into the transaction-private write set and returns. Otherwise, there is a conflict and the default contention management policy is to immediately abort the transaction (we will show how one can change this behavior to provide fairness shortly).

If the lock bit is not set, the transaction tries to acquire the lock using a CAS operation. Failure indicates that another transaction has acquired the lock in the meantime and the whole procedure is restarted. If the CAS succeeds, the transaction becomes the owner of the lock. This basic design thus implements visible writes with objects being acquired when they are first encountered.

When reading a memory location, a transaction must verify that the lock is neither owned nor updated concurrently. To that end, the transaction reads the lock, then the memory location, and finally the lock again (obviously, appropriate memory barriers are used to ensure correct ordering of accesses). If the lock is not owned and its value (i.e., version number) did not change between both reads, then the value read is consistent. If the lock is owned by the transaction itself, the transaction returns the value from its write set. Once a value has been read, LSA checks if it can be used to construct a consistent snapshot. If that is not the case and the snapshot cannot be extended, the transaction aborts.

Upon commit, an update transaction that has a valid snapshot acquires a unique commit timestamp from the shared clock, writes its changes to memory, and releases the locks (by storing its commit timestamp as version number and clearing the lock bit). Upon abort, it simply releases any lock it has previously acquired. Refer to [RFF06] for more details about the LSA algorithm.
3.4 Design and implementation

3.4.3 PROGRESS AND FAIRNESS

An important observation is that the basic TiNYSTM algorithm does not provide liveness guarantees even when considering only well-behaved transactions. In particular, a set of transactions can repeatedly abort each other, thus creating livelocks. Furthermore, there is no fairness between transactions: a long-running transaction might be taken over and aborted many times by shorter update transactions, in particular if the former performs numerous invisible reads.

To address these problems, we introduce two mechanisms that make up Mode 2. The first one consists of introducing “visible reads” after a transaction has aborted a given number of times because of failed validation (i.e., due to invisible reads). To that end, in addition to the WR bit used for writers, we use an additional RD bit in the lock metadata to indicate that a transaction is reading the associated data (see Figure 3.2). Using a different bit for visible readers allows more concurrency because an invisible reader is still allowed to read data that is locked in read mode. Other conflicts with visible readers are handled as for writers, i.e., only one transaction is allowed to proceed. The use of visible reads makes all conflicts detectable at the time data is accessed: a well-behaved transaction that wins all conflicts is guaranteed not to abort.

This mechanism alone is not sufficient to guarantee neither progress nor fairness. Depending on the contention management strategy, transactions can repeatedly abort each other, or a transaction might always lose to others and never commit. To address the fairness problem, we need to be able to prioritize transactions and choose which one to abort upon conflict. That way, we can ensure that the transaction with the highest priority level wins all its conflicts.

A transaction that cannot commit in Mode 1, first switches to visible reads. If it still fails to commit after a given number of retries with visible reads enabled, it tries to enter a privileged priority level that accepts only one thread at a time. Entry into this priority level is guarded using Lamport’s bakery algorithm [Lam74] that provides fairness by granting permission in the order in which transactions try to acquire the bakery lock. Because the number of steps that transactions are allowed to execute with priority is limited (see Section 3.4.6), each acquire attempt will finish in a finite number of steps. The privileged thread can steal a lock from its current owner by atomic-or’ing the PR bit to 1 before acquiring it. The bit indicates that a transaction is about to steal the lock (see Figure 3.2). As explained in Section 3.3, this will ensure that any other thread attempting to CAS the lock metadata will fail (because it expects the PR bit to be cleared), while the privileged thread will succeed.
3.4.4 SAFE LOCK STEALING

Due to the lock-based nature of our base STM, being able to safely steal locks from transactions is necessary to build a robust STM. Our system model eases this because it requires that STM code is well-behaved and only application code can crash or loop infinitely often.

To understand how lock stealing works, consider Figure 3.3 that shows the different states a transaction can take. The normal path of a transaction is through states IDLE, ACTIVE (when transaction has started), VALIDATE (upon validation when entering commit phase), and COMMIT (after successful validation when releasing locks). A transaction can abort itself upon conflict (from ACTIVE state), or when validation fails (from VALIDATE state).

An active transaction can also be forcefully aborted (or killed) by another transaction in privileged priority (dashed arrow in the figure). This happens when the privileged transaction $tx$ wants to acquire a lock that is already owned, i.e., with the RD or WR bit set. In that case, $tx$ first reserves this lock for the privileged transaction by atomic-or-ing the PR bit to 1. This wait-free operation also ensures that other non-privileged transactions will notice the presence of $tx$ and will not be able to acquire the lock or clear the PR bit anymore (see Figure 3.2). In ROBUSTM, all lock acquire and release operations must be performed using CAS, which will fail for non-privileged transactions if the PR bit is set.

After reserving the lock, $tx$ can continue with actually stealing the lock. It loads the value of the lock again and determines whether there was an owner transaction. If so and if the owner is in the IDLE state, it can just acquire the lock. If the owner is in the VALIDATE or COMMIT states, $tx$ waits for the owner to either abort (e.g., because validation failed) or finish committing. We do not abort validating transactions because they might be close to successfully committing. Because we assume that STM code is well-behaved and because read sets are finite, commit attempts execute in a finite number of steps. Note that a successfully committed transaction releases only the locks whose PR bit is not set. This process works as long as there is at most one transaction in the privileged priority level that can steal locks.

If the owner transaction is in ACTIVE state, $tx$ attempts to abort the owner by using CAS to change the state to ABORT. After that or if the owner is already in state ABORT, $tx$ acquires the lock using CAS but while doing so expects the value that the lock had after the atomic-or. The PR bit is only used during lock stealing and is not set after $tx$ acquired the lock. Transactions check whether they have been aborted within each STM operation (e.g., loads). Note that a transaction’s state is versioned to avoid ABA issues on lock owners, i.e., $tx$ can distinguish if the transaction that previously owned the lock aborted and retried while performing the lock stealing.

Figure 3.3: States during the lifetime of a transaction.
3.4 Design and implementation

3.4.5 DEALING WITH CRASHED TRANSACTIONS

Using a traditional lock-based STM could lead to infinite delays in case a thread that has acquired some locks crashes. Because ROBuSTM supports lock stealing, the crash of a transaction that is not in privileged priority level and that is not in the COMMIT state does not prevent other transactions from safe lock stealing introduced in Section 3.4.4.

ROBuSTM makes use of the crash detector included in the MSM to deal with crashed transactions. In practice, events that cause a thread to crash (e.g., a segmentation fault or an illegal instruction) are detected by the operating system and a thread can request to be notified about such events by registering a signal handler. If a signal is received by a thread that indicates a crash, the thread will abort itself if it is in the ACTIVE state to speed up future acquisitions by other threads. If the thread is in the COMMIT state and already started writing its modifications to memory, it will finish committing. The intention there is to always keep the shared state consistent and to reduce the contention on locks. Transactions in privileged priority level that encountered a thread crash additional release their priority.

3.4.6 DEALING WITH NON-TERMINATING TRANSACTIONS

The main problem that we face when designing a robust STM is how to deal with non-terminating transactions as the locks they hold can prevent other transactions from making progress. Two different kinds of non-terminating transactions have to be distinguished: (1) transactions that are in ACTIVE state but stopped executing STM operations, and (2) transactions that still perform STM operations (e.g., in an infinite loop). Both correspond to non-crashed threads and never reach the VALIDATE state.

Let us first consider how ROBuSTM handles threads that stopped executing STM operations (e.g., the thread is stuck in an endless loop). In the simplest case, the thread did not acquire any locks and thus does not prevent other threads from making progress and can be tolerated by the system. If the non-terminating transaction already acquired locks, it may run into a conflict with another thread. Eventually, the conflicting thread will reach the privileged priority level and again run into a conflict with the non-terminating transaction. It will then force the non-terminating transaction to abort and steals the lock. Since the status of a thread is only checked during STM operations, the non-terminating transaction will not discover the update and will remain in the ABORT state. Other transactions that encounter a conflict with a transaction in ABORT state can simply steal the lock.

A non-terminating transaction that still performs STM operations will discover the update of its state to ABORT. It will roll back and retry its execution. If, during the retry, it becomes again a non-terminating transaction that owns locks, it will be killed and retried again. It can therefore enter the privileged priority level and still behave as a non-terminating transaction, hence preventing all other transactions from making progress because it wins all conflicts. Since we assume that the state of a computer is finite, for each well-behaved and privileged transaction \( tx \) there exists a maximum number of steps, \( \text{maxSteps} \), such that \( tx \) will execute at most \( \text{maxSteps} \) before trying to commit. \( \text{maxSteps} \) is not known a priori and hence, we cannot reasonably bound the number of steps that a privileged transaction is permitted to execute without risking to prevent some well-behaved transactions from committing.

MSM permits us to deal with non-terminating transactions running at the privileged priority level as follows. The privileged thread \( th \) receives a budget of at most a finite number of steps but at least \( \text{quantum} \) steps, where \( \text{quantum} \) is a dynamically updated value. Initially, \( \text{quantum} \) is set to some arbitrary value that we assume to be smaller than \( \text{maxSteps} \). The privileged thread \( th \) is forced to the ABORT state after \( \text{quantum} \) steps (determined with the help of the performance counters) and is removed from the privileged priority level.

If the formerly privileged transaction \( th \) notices that it has been aborted and exceeded its quantum, it clones its thread (illustrated in Figure 3.4). The clone consists of a separate
address space that is copied on write from the parent and a single thread that runs in isolation. Transactional meta data of all threads in the parent is copied with the address space. The clone then continues to execute the transaction in a checker run using the meta data to resolve conflicts. There are two cases to consider. (1) If \( t_h \) is well-behaved, it will terminate after running for, say, \( \text{childSteps} \). At this point, the child will return \textit{success} and the parent thread will increase \textit{quantum} by setting it to a value of at least \( \text{childSteps} \). Then, the parent thread will re-execute \( t_h \) at privileged priority with at least \( \text{quantum} \) steps. If the new \( \text{quantum} \) was not sufficient, e.g., because of non-determinism, it will be increased iteratively. (2) If \( t_h \) is not well-behaved, it will not terminate the checker run. In this case, the parent thread will wait forever for the child thread to terminate. Because the parent thread has aborted, it will not prevent any of the well-behaved threads from making progress.

3.5 EVALUATION

In this section, we evaluate the performance of ROBUSTM. We are specifically interested in showing that (1) it provides high throughput in good cases with little contention, (2) it provides fairness by guaranteeing progress of individual transactions, and (3) it tolerates crashed and non-terminating transactions.

We compare ROBUSTM against four state-of-the-art STM implementations: a C++ implementation of TinySTM [FFR08]; TinyETL, a C++ implementation of the encounter-time locking variant of TinySTM; TL2 [DSS06], an STM implementation that uses commit-time locking; and NB STM [MM08a], which combines efficient features of lock-based STM implementations with a non-blocking design, as our algorithm does. The NB STM implementation that we use is a port of the original SPARC implementation to the x86 architecture.

For our evaluation, we use well-known micro-benchmarks as well as applications of the STAMP [Cao+08] benchmark suite. The \textit{intset} micro-benchmarks perform queries and updates on integer sets implemented as red-black tree and linked list. We use the \textit{bank} micro-benchmark to evaluate fairness: some threads perform money transfers (i.e., one withdrawal followed by a deposit) concurrently with long read-only transactions that compute the aggregated balance of all accounts. From the STAMP benchmark suite [Cao+08] we chose \textit{Vacation}, \textit{KMeans} and \textit{Genome}. Vacation emulates a travel reservation system, reading and writing different tables that are implemented as red-black trees. KMeans clusters a set of points in parallel. Genome performs gene sequencing using hash sets and string search.

Our tests have been carried out on a dual-socket server with two Intel quad-cores (Intel XEON Clovertown, executing 64-bit Linux 2.6). We compiled all micro-benchmarks using the

![Figure 3.4: Illustration of the checker run that clones the thread to run in isolation.](image)
Dresden TM Compiler [Chr+10], which parses and transforms C/C++ transaction statements and redirects memory accesses to an STM.

### 3.5.1 THROUGHPUT FOR WELL-BEHAVED TRANSACTIONS

We first evaluate transaction throughput for lock-based and non-blocking STM implementations. There are no crashes or non-terminating transactions present.

Figure 3.5 shows the bank benchmark with low load under different STM runtimes. The left and middle plots show throughput for both transfer and aggregate-balance transactions. The lock-based STMs perform significantly faster than NB STM because the chosen non-blocking STM still requires an indirection step in case of contention. These results show why we would like ROBUSTM to perform as well as blocking STMs. ROBUSTM has more runtime overhead than TINYETL and TINYSTM but is on par with TL2. Figure 3.6 shows performance results for additional micro-benchmarks and STAMP applications. Results for NB STM are only presented for the red-black tree because it requires manual instrumentation and it is not supported by the STAMP distribution. These results are in line with the bank benchmark results, showing that TINYSTM and TINYETL perform best, followed by ROBUSTM, then TL2 and finally NB STM.

The right plot of Figure 3.5 shows that the fairness that ROBUSTM helps avoid starvation of the long aggregate-balance transactions with visible reads. In this plot, we only show the throughput of a single thread that is performing aggregate-balance (read-all) transactions. The guarantee for individual threads to make progress under MSM provides fairness for transactions that otherwise would not have a good chance to commit. Other STMs with invisible reads that simply abort upon conflict do not perform well because the read-only transaction will be continuously aborted.
3.5.2 TOLERATING CRASHES AND NON-TERMINATING TRANSACTIONS

We now evaluate transaction throughput in the presence of crashes or non-terminating transactions. Ill-behaved transactions are simulated by injecting faults at the end of a transaction that performed write operations (i.e., it holds locks). We inject thread crashes by raising a signal and simulate non-terminating transactions by entering an infinite loop. The infinite loop either performs no operations on shared memory or continuously executes STM operations (e.g., transactional loads).

Orthogonal to the robustness that ROBUSTM offers for synchronization, applications must be tolerant against faults of its threads. During the setup of our experiments we discovered two major problems with the thread-based benchmarks. (1) Barriers must be tolerant to faulty threads that never reach the barrier because of a crash or non-terminating code. (2) The workload cannot be pre-partitioned to the initial number of threads. Instead, it must be assigned dynamically, e.g., in each loop iteration. Therefore, we chose only a selection of STAMP applications that could be easily adapted. Using ROBUSTM, an adapted application with initially \( N \) threads can tolerate up to \( N - 1 \) faults because even ill-behaved transactions prevent the thread from processing its work. Increasing the number of tolerated faults would require a change in the programming model, e.g., based on a thread pool, and is not in the scope of this chapter.

Figure 3.7 shows the performance of ROBUSTM compared to TINYETL, the most efficient STM in the previous measurements. In each plot of the figure, we show the performance when some threads are faulty in the baseline 8-thread run. TINYETL is a run where faulty threads are simply not started in the runs, and thus shows the baseline. “Well-behaved” is similar (only well-behaved transactions), but uses ROBUSTM. The other three lines show the performance in the presence of transactions that are not well-behaved. Faults were injected as early as possible, except for Genome, where they were injected in the last phase and can only be compared to the 8-thread runs. The results show that ROBUSTM can ensure progress for an increasing number of injected faults. In fact, it can even compete with the throughput of the “well-behaved” case for the considered benchmarks.

To illustrate how ROBUSTM behaves when non-terminating transactions are present, Figure 3.8 shows the number of commits and aborts over periods of time for the red-black tree benchmark. In the left graph, the benchmark is executed with two threads and one transac-
tion enters an infinite loop that does not call STM operations. The remaining thread runs into a conflict and aborts repeatedly until it enters the privileged priority level. It then is allowed to kill the non-terminating transaction in order to steal its locks. Afterwards, the throughput picks up to the level of a single threaded execution. The right graph shows a scenario taken from Figure 3.7 with eight threads and one transaction that enters an infinite loop with STM operations. All remaining seven threads abort after running into a conflict and eventually reach privileged priority. Because the non-terminating transaction detects that it has been aborted during its STM operations, it retries. Thus, it must be aborted multiple times until it gains privileged priority. While the non-terminating transaction executes privileged, other threads wait and check the quantum of the non-terminating transaction. After the transaction detects that it was aborted because its quantum expired, it will clone its thread to enter the checker run. The period between gaining privileged priority and entering the checker run is much longer than the allowed quantum because it includes the costly clone of the process. After the initialization of the checker is finished, all locks are released in the parent process and the other threads can continue.

The results show that despite several crashed or non-terminating threads, ROBUSTM is able to maintain a good level of commit throughput, effectively shielding other threads from failed transactions. We tested injecting faults in other STM implementations to justify our design decisions. Lock-based designs that acquire locks at commit-time (e.g., TL2) seem promising towards tolerating crashes and non-terminating transactions. Problems arise when fairness is at stake because memory accesses cannot be easily made visible. For implementations with encounter-time locking that simply abort on conflict (e.g., TINYETL), transactions that are not well-behaved or own locks lead to deadlocks. To overcome deadlocks, lock stealing and external abort of transactions must be supported. This will allow to tolerate crashes but not non-terminating transactions as they might continuously retry. We found that none of further existing approaches for contention management (see Section 3.2) met our robustness requirements.

3.6 SUMMARY

Robustness of transactional memory has often been ignored in previous research as the main focus was on providing performance. Yet, robustness to software bugs and application failures is an important property if one wants to use transactional memory in large mission-critical or safety-critical systems.

In this chapter, we have introduced the multicore system model (MSM) that is practical in the sense that it reflects the properties of today’s multicore computers. We have shown that (1) it is possible to build a robust STM with performance comparable to that of non-robust
state-of-the-art STMs, and (2) we can implement such an STM under MSM.

Our experimental evaluation indicates that robustness only has a small additional overhead in the good case (i.e., no or few ill-behaved transactions), and performance remains good even when there are crashed and non-terminating threads. We expect to further improve efficiency by tuning the configuration parameters at runtime. For ROBUSTM, these are especially the number of retries (1) after which transactions switch to using visible reads and (2) after which they attempt to run as a privileged transaction. Previous work has shown this to be very beneficial in the case of other STM configuration parameters [FFR08]. We also expect that pairing this work with operating-system scheduling [Mal+10] could enable interesting optimizations.
4 TRANSACTIONAL ENCODING FOR TOLERATING TRANSIENT HARDWARE ERRORS*

*The contents of this chapter first appeared at SSS ’13 [Wam+13b].
4.1 Introduction

The decreasing feature size of integrated circuits leads to less reliable hardware with higher likelihood for errors. Without adding additional failure detection and masking mechanisms, the next generations of CPUs would at least be unfit for executing mission- and safety-critical applications. One common approach is the replicated execution of programs on redundant cores, which is increasingly difficult considering that most programs are non-deterministic. To be able to detect and mask execution errors, one typically needs to execute three copies of each thread.

In this chapter, we propose and evaluate transactional encoding, a novel approach to detect and mask transient hardware errors such that one can build safe applications on top of unreliable components. Transactional encoding relies on a combination of arithmetic codes for detecting transient hardware errors and transactional memory for recovery and tolerance of transient errors. We present a prototype software implementation that encodes applications using an LLVM-based compiler and executes them with a customized software transactional memory algorithm. Our evaluation shows that our system can successfully detect and mask between 59-79% of transient hardware errors. The main objective of our prototype is to improve the detection and masking of transient errors by an ongoing hardware implementation of a resilient processor, that will reduce the current overhead substantially.

4.1 Introduction

The dependability of hardware components in a computing systems is influenced by several factors. Some are related to the environment (e.g., system operating in tough conditions such as in space or at extreme temperatures) while others are driven by the evolution of technology. Notably, the increase in transistor density of integrated circuits leads to less reliable hardware and higher likelihood for transient errors [Bor05]. Recent research has also shown that significant energy savings can be achieved by operating at lower, almost unsafe voltage levels, albeit at the price of increased error rates (e.g., [Ern03; Rob+05]). Such transient hardware errors are particularly difficult to handle as they cannot be detected easily.

Traditionally, wrong executions of programs are detected by means of redundant executions and comparison of the results. Redundant execution is effective under the assumption that the program is deterministic, i.e., the result of a computation only depends on its input. However, most non-trivial applications nowadays are non-deterministic, e.g., because of concurrency or errors returned by some replicated system calls. This non-determinism makes replicated execution challenging because one must rely not only on synchronization of the input but also of thread scheduling and system calls. Moreover, to be able to not only detect but also to mask transient errors, one needs triple executions and voting.

In this chapter, we tackle the problem of building possibly non-deterministic software systems that can tolerate a large fraction of transient execution errors. Indeed, safety critical systems require that, depending on the safety integrity level, a specified fraction of failures do not result in a safety violation. By tolerating a large fraction of failures, hardware consisting of unreliable components can be used in mission and safety-critical applications.

The underlying idea of our approach is to combine two techniques: (1) encoded processing [For89], which provides means to detect incorrect execution of code and guarantees data integrity, and (2) transactional memory (TM) [HLR10], which supports speculative execution of code and provides checkpoint/rollback mechanisms to restart erroneous operations. This novel combination of techniques allows us to detect and tolerate a significant fraction of transient errors such as data corruptions or execution errors. We show a proof-of-concept implementation that can detect and recover from a wide range of errors, albeit with some runtime overhead resulting from the software-only nature of our prototype. Note that our focus in this study is not on performance as there are ongoing efforts to put these mechanisms in hardware.
The chapter makes the following contributions: (1) We introduce a novel approach to detect and tolerate transient errors when executing applications on unreliable hardware (Section 4.3). Error detection is achieved using encoded processing and symptom-based error detection. Fault tolerance is supported by TM, providing means to recover to a correct state. (2) We automatically transform and instrument applications written in C using an encoding compiler (Section 4.4). This allows to apply our approach to existing applications without manual adaption of memory accesses. (3) We base the checkpointing mechanism on TM that we streamlined for high performance failure atomicity (Section 4.5). The TM selectively replicates memory so that the runtime system can tolerate errors that are not recoverable by a rollback. The replica is used to check and correct the consistency of the memory without aborting the transaction. (4) We present a prototype implemented in software and study the design of our self-healing approach. We apply the prototype to several example applications and evaluate the effectiveness of the aspired error tolerance and its resource overhead (Section 4.6). Results indicate that we can tolerate up to 59-79% of transient errors.

4.2 RELATED WORK

Dependable mainframe systems typically add fault tolerance by introducing redundancy at different levels in hardware [HHJ90; Sle+99]. Faults are mitigated by combining information redundancy (e.g., checksums) and redundant execution, either using replicated hardware components or sequential re-execution over time.

Error detection and correction (ECC) uses information redundancy in form of parity data. Current ECC hardware implementations correct single event upsets that result in a single bit flip and detect double bit flips. However, ECC is applied to register files or ALU circuits only in custom processors and causes a high space and computing overhead.

Combinational logic within processors can be protected by either hardware-based [WB91] or software-based [OMM02; OSM02] duplication. A common hardware approach uses multiple identical lock-stepped processors to run copies of the same program in parallel [FT11; IBM08; Ng07]. In each cycle, their state should be identical since the same inputs are provided. The output state is compared using a voter. In a dual modular redundant (DMR) setup, errors can only be detected or, if in a master/slave configuration, the slave can continue from the last known valid state [Ber+05]. Using triple modular redundancy (TMR), the voter can detect the erroneous state and discard it [Yeh01]. The high overhead from required redundant computation can be reduced by verifying only the integrity of the core processor’s computation [Aus99] or checking only invariants [MBS07].

Custom hardware solutions are often too costly for general use. Instead, software-only techniques achieve reliability using unreliable commodity hardware and redundancy on the application-level. For example, SWIFT [Rei+05] is an instrumentation technique that computes in software duplicate versions of all register values and inserts validation instructions before control flow and memory operations. With the lack of duplicated memory operations, no end-to-end detection of hardware faults is provided. SWIFT-R [CRA06] adds majority voting before critical instructions and allows recovery. ReStore [WP06] detects failures by its symptoms (e.g., exceptions) but sacrifices error coverage if no symptom gets activated. Li et al. [Li+08] enhance the symptom model with operating system support. DieHard [BZ06] uses probabilistic memory safety based on randomization and process replication to overcome memory errors.

Error recovery techniques are generally based on checkpoint/rollback mechanisms [LAK09; Nak+09; Sor+02] and can be triggered by error detection [Rin+04]. Checkpointing alone has the drawback that it can get corrupted by hardware errors and does not allow a selective repair but only a rollback. Samurai [PGZ08] is a robust runtime system that protects critical memory without rollback but forward recovery, i.e., fixing the memory using the replica, but requires
4.3 System Design

TM [HLR10] provides an automated form of checkpoint/rollback. It supports speculative execution of code and is originally used as a synchronization mechanism to provide an alternative for locks, aiming to simplify parallel programming [Fel+10; FFR08; Spe+06]. It has been argued [Cri+13; FW11], however, that in the context of embedded systems TM should be limited to achieve failure control rather than concurrency control, and thus to provide a lightweight recovery control mechanism. SymptomTM provides a recovery strategy based on the abort operation of hardware TM, triggered by symptom-based error detection [Yal+11]. FaultTM uses TM for error detection and recovery [YUC13]. The detection is based on a redundant execution of the application at thread-level and checks that the update logs of replicated transactions match upon commit. It requires twice the number of processors plus additional memory for the replicated logs.

4.3 SYSTEM DESIGN

Our goal is to allow applications to execute in a fault tolerant manner in an environment that is prone to transient errors. Traditionally, such dependability is achieved by replicating the entire execution of the application, either in time or space. Compared to existing approaches (see Section 4.2), we want to minimize the replication overhead. Our approach is to replicate the application’s state in memory only, and to add parity information for operations. The latter is used for error detection and the former for error recovery.

4.3.1 SYSTEM MODEL

The system model captures the properties of commodity hardware, the application, and the errors that can occur. Commodity hardware is composed of unreliable off-the-shelf components. It does not provide hardware support for error detection or correction in any of its components.

The hardware executes operations using the CPU. It consists of processor cores with registers and attached memory. The registers and memory hold the state of the application and are connected by bus or interconnect networks. The operations access and modify the state. Each operation consists of an instruction with operands as parameters, to which it applies an operator. Operations can have at most one memory operand.

The hardware can suffer from transient faults that are bound in time, i.e., if the operation is repeated the resulting transient error will not re-occur, but may result in incorrect operation execution by altering state, operands or operators. Typical examples are bit flips due to radiation or noise from the power supply.

Following the error model of Forin [For89], the system can suffer on the software-level from the following symptoms caused by hardware errors: (1) a modified operand with a transient error read from the state; (2) a faulty operation uses correct operands but a corrupted operator produces incorrect state; (3) an exchanged operand is executed by the operation, e.g., after a fault on an address line; (4) an exchanged operator is executed with correct operands, e.g., addition instead of subtraction; or (5) a lost update does not manifest the operation’s result in the state, i.e., the state is not up-to-date. The symptoms can be combined with one another to represent other symptoms using the error model.

Error detection allows us to identify activated hardware errors before they propagate to a failure. The correction of an error requires recovery in order to revert the system to a correct state. The tolerance of transient errors is based on a combination of detection and recovery. It can selectively correct and decide on a re-execution of operations. If tolerance fails, the error is considered permanent.
4.3.2 DESIGN OVERVIEW

Our approach of transactional encoding has the objective to enable applications to tolerate transient hardware errors. The design is based on a combination of arithmetic codes for error detection and transactional memory for error recovery. We extend the system by a runtime that allows us to tolerate unrecoverable errors by selectively correcting the state. Figure 4.1 depicts an overview of the transactional encoding process.

Arithmetic codes provide end-to-end error detection and are implemented in software, so all unreliable components are covered. This is achieved by adding redundancy to the data, so that one can detect errors that affect data during storage, transport, or operation, according to the error model (see Section 4.4).

Transactional memory (TM) continuously captures in a log all state changes performed by the operations of a transaction. Thus, TM provides a straightforward check-pointing technique that we use for error recovery. If an error is detected during the execution of an operation, the transaction will be aborted and all changes will be rolled back using the log, thus returning to the state prior to the start of the transaction. This property is called failure atomicity. If the transaction reaches its end, it commits and makes all state changes permanent.

No guarantee is given that the application returns to a valid state after an abort. Therefore, TM additionally maintains replicas of all memory update operations in the log. The replicas allow us to detect corruptions not discoverable by arithmetic codes and to tolerate and repair corrupted state during the re-execution of the transaction (see Section 4.5).

4.4 ERROR DETECTION

Our error detection is based upon encoded processing, which applies arithmetic codes automatically to C code using an encoding compiler. It validates the correctness of the operand’s state output and activates the error tolerance mechanisms if an inconsistency is detected.

4.4.1 ENCODED PROCESSING

Encoded processing adds redundancy to any value that is part of an application’s state. This redundancy transforms the original domain of values into a larger domain where only a small subset of values are valid code words. Figure 4.2 shows the relation between valid code words and all possible code words. Correctly-executed arithmetic operations on code words
4.4 Error Detection

Figure 4.2: Code word domain of arithmetic codes and possible computation [Sch11].

As seen in Figure 4.2, valid code words are represented by the filled circle, and all possible code words are represented by the open circle. Fault-free addition is denoted by a plus symbol, and faulty addition is denoted by a cross symbol. Read/write code words are represented by a red arrow, and read/write non-code words are represented by a blue arrow.

We preserve the code (case 1). However, a hardware error affecting the computation, e.g., a bit-flipped operand (case 2) or an erroneous arithmetic operation (case 3), results with a high probability in an invalid code word. Additionally, encoded processing allows us to detect errors during transport or storage of values, as they most likely also destroy the code word property.

For adding the redundancy, the encoding compiler uses arithmetic error detection codes. Well known codes comprise:

1. **AN**: For AN codes, the set of valid code words comprises the integral multiples of a compile-time constant $A$. Consequently, the code only allows for detection of value errors such as modified operands and faulty operations according to the error symptom model (see Section 4.3).

2. **ANB**: If the control flow gets modified, the processor might (a) use a different operand than the intended one (exchanged operand), or (b) perform a different computation (exchanged operator). Therefore, Forin [For89] introduced value-independent static signatures, such that the resulting signature depends on both the correct operand signatures and operator.

3. **ANBDmem**: In order to detect the utilization of outdated values (lost update), e.g., due to address bus errors, Forin [For89] also introduced a generic timestamp for all values (ANBD code). In contrast, the encoding compiler only applies timestamps to values stored in memory but not processor-internal registers (ANBDmem code) [Sch+10a].

Measurements show that the increased detection capability of ANB and ANBDmem codes comes at the expense of more processing overhead [Sch+10c]. In our prototype, we use AN codes because it is the only encoding currently supported by the compiler and combine it with symptom-based error detection that identifies anomalous application behavior (e.g., crashes) [Li+08; WP06]. We extend the transactional memory with masking of values to protect from an exchanged operand, and we validate the transaction at commit to discover lost updates (see Section 4.5). This allows us to cover symptoms not detectable by AN codes.

### 4.4.2 APPLICATION ENCODING

The encoding compiler is a C source-to-source compiler, i.e., it processes C code as input and generates C code as output [Süß+11]. Doing so has the advantage of being able to (a) be used in existing tool chains in front of a target C compiler, and (b) support a whole range of target platforms even though LLVM provides no backend. It consists of two major components as part of Figure 4.1:

1. **Encoded Operations**: A library of encoded operations, which provide encoded variants of all operations present in the original application and floating point arithmetics [WF07].
2. **Transformer:** The transformer operates on LLVM intermediate bitcode [LA04] replacing each original operation (e.g., arithmetic, logical, address computations) with an appropriate encoded one.

As some parts of applications are usually more safety-critical than others, the encoding compiler can adapt the scope of protection at a fine granularity by only encoding selected modules. Algorithm 4.1 shows the transformer output for an example application originally comprising two modules. The main module (lines 1–3) initializes the application and is not encoded. The encoded module (lines 11–16) contains the safety-critical algorithm, i.e., a counter using a global variable as its state. The encoded `INCCOUNTER_e` function loads the current value, adds an increment, and returns the stored result.

Since the interfaces of unencoded and encoded functions have different semantics, the encoding compiler additionally generates wrappers for public functions (lines 4–10). Those *public wrappers* (1) encode their parameters, (2) call their encoded counterpart, and (3) optionally decode the return value (lines 6–8). Similarly, if encoded code attempts to call external functions (e.g., library or system calls), the transformer generates *external wrappers* working in the opposite direction. For combining encoded processing and TM, the wrappers also contain transaction demarcations (lines 5 and 9).

Thus, any state being externalized is decoded and checked, allowing the runtime to take recovery actions if necessary (see Section 4.5). Note, to avoid expensive checks of intermediate values, our approach defers code word checks to the latest possible point by relying on the error propagation of the employed arithmetic code. Hence, the system does not fail fast, but optimistically executes until the end of the transaction based on the assumption that errors occur rarely. Consequently, it requires means to recover even if the fault leading to an error corrupted the application state in a previous transaction.

### 4.5 ERROR RECOVERY AND TOLERANCE

The detection of an error is a violation of the application’s integrity and triggers the recovery. Error recovery and tolerance are implemented by TM, which is traditionally used for concurrency control. However, here the focus is on dependability and we use a streamlined TM variant that does not suffer from overheads introduced by synchronization capabilities. The TM supports two modes: The *fast mode* is optimized for high throughput in the good case when no errors occur. Error detection is lazy, i.e., only at commit, and TM is used for failure atomicity. The TM performs backward recovery and the transaction is restarted from its beginning. The *self-healing mode* implies more checks and can fix the state selectively during execution instead of aborting the transaction. It is enabled upon retry of the transaction and performs eager error detection, i.e., on each state access. Upon error detection, the replica is used for forward recovery.

The TM is integrated with AN encoding such that an abort will be requested in case a non-valid code word was detected in an operand. In order to support a rollback, a valid checkpoint is required, which is built incrementally by recording all write operations that update the state. If a recovery based on rollback is not successful, the TM will identify the location of the state inconsistency and replace it with the most recent correct version from the state in the checkpoint.

#### 4.5.1 FAILURE ATOMICITY

The TM runtime library is called from the encoded operations (see Algorithm 4.1). Transactions are started in the generated wrapper of an encoded function invocation and committed when it returns (lines 5 and 9). Within the transaction’s boundaries, other functions can be called, but no state can be made externally visible. Native functions (not encoded) are not executed
within transactions, thus, no recovery is provided. This also resolves irrevocability constraints known for TM from concurrency control [BLM05] because system calls and calls to external code will be executed outside the transaction (line 3 causes an irrevocable system call). All accesses to the state are redirected to the TM by invoking read and write functions from the encoded operations (lines 13 and 15).

Update operations of the state will not only be reflected by the current state in memory but also by a history at a different memory location. The history is maintained by the TM and is called write-set (see Figure 4.3). One entry is attached per update operation. A memory location of the state is identified using a decoded address and has an encoded value. Each write-set entry consists of the encoded address, the current encoded value of the memory at the time the entry was created, and the memory’s previous encoded value. Note that we replicate (1) the value in memory as current value in the entry, and (2) the historic value as old value (e.g., $v_{a3e}$ is overwritten for $a3e$).

The write-set maintains a history of all executed transactions and their entries. The current transaction’s subset is identified using indices: $startIdx$ points to the first entry and $currIdx$ points to the next free entry after the subset. We assume for our prototype that sufficient memory is available for an infinite history that contains all values ever written during the application’s execution.1

We now explain the implementation of failure atomicity using the TM in Algorithm 4.2. With each switch from an unencoded to an encoded module, a new transaction is started. At the START of a transaction, we enable the fast mode and store the current register state (lines 8–9).

The WRITE function is called when an operation updates the memory state. It takes as parameters the encoded address and the encoded operand’s value. We check if the value is a VALID operand (lines 22–23) to protect the write-set replica of the memory. An invalid currently value in the write-set would prevent our self-healing after the transaction committed. We must DECODE the address (line 24) in order to find the actual location in the state. If the address is a valid code word, its decoded value is returned, otherwise the transaction is aborted (line 3–6). The encoded address, its dereferenced current value, and the new encoded value are appended to the write-set and the index that points to the current write-set entry is advanced (lines 25–26). Finally, the memory state is updated with the new encoded value at the decoded address (line 27).

The READ function must also DECODE the address (line 13), but in fast mode it simply

---

1Only the most recently used entry for each encoded address plus the entries of the current transaction are required in the write-set for our approach. All other entries are considered outdated and can be garbage collected, which is not yet implemented.
Algorithm 4.1: Counter application.

```
1 function MAIN // Main function module
2     c ← INCCOUNTER(42)
3     PRINT(c)
4 function INCCOUNTER(inc) // Wrapper function
5     START
6     inc_e ← ENCODE(inc)
7     c_e ← INCCOUNTER_e(inc_e)
8     c ← DECODE(c_e)
9     COMMIT
10    return c
11
12 function INCCOUNTER_e(inc_e) // Encoded module
13     c_e ← READ(countere)
14     c_e ← c_e + e inc_e
15     WRITE(countere, c_e)
16    return c_e
```

returns the encoded value from the memory address without checking if it is a valid code word (lines 14–15).

Transactions reach their COMMIT if no errors were detected during their execution. Only the return value of the encoded function is exposed from the state, so only this value is validated during the DECODE before the COMMIT (see Algorithm 4.1, line 8 and Algorithm 4.2, line 3). The TM prepares for the next transaction by advancing the write-set index startIdx to currIdx (line 11).

The error detection mechanisms can trigger an ABORT of a transaction. All changes to the state made by the current transaction must be rolled back. This is done by applying the old encoded values of the write set (see Figure 4.3) in reverse order to the state (line 32). We enable the self-healing mode and reset currIdx back to the startIdx of the write-set (lines 31, 33). Finally, the transaction is restarted by resetting the register state.

4.5.2 SELF-HEALING

The error tolerance of transactional encoding is based on the assumption that transient errors will not occur again when the operation is repeated. Abort and retry will re-execute the operation, but there is no guarantee that the aborted transaction in fast mode has rolled back to a valid state. ABORT (see Algorithm 4.2, line 32) does not validate the old values from the write-set written back to the state because the initial values of the state might not be valid code words. Any invalid values restored by ABORT will not be externally visible. Upon next access of the invalid state, we use the replicated state information from the write-set to heal the state during transaction re-execution.

The self-healing mode is enabled after an ABORT in the fast mode. We add three extensions for error tolerance: (1) validation can identify inconsistencies between the state and the write-set; (2) additional masking of values enhances the encoding semantically and can detect if a value belongs to a given address; and (3) signal handling enables the recovery from errors that were not immediately identified.

Table 4.1 summarizes which self-healing mechanism will be activated on different kinds of state corruptions. Three outcomes are possible: (1) the ABORT fixed the state based on failure atomicity, (2) the state is corrected during a READ, or (3) KILL application if the state cannot be corrected or an invalid address was encountered during a re-execution.
Algorithm 4.2: TM for dependability.

1. \( \text{currIdx} \leftarrow \text{startIdx} \leftarrow 0 \)  // Initialize write-set indices
2. function \( \text{DDECODE}(\text{val}_\text{e}) \)  // Check and decode word
   3. if \( \text{VALID}(\text{val}_\text{e}) \) then
      4. \( \text{return} \ \text{GET}(\text{val}_\text{e}) \)  // Return decoded value
   5. else
      6. \( \text{ABORT} \)  // Abort transaction
3. function \( \text{START} \)  // Begin or restart a transaction
   8. \( \text{fastMode} \leftarrow \text{true} \)
   9. \( \text{setjmp}(\text{ctxt}) \)  // Store the register state
10. function \( \text{COMMIT} \)  // Commit a transaction
   11. \( \text{startIdx} \leftarrow \text{currIdx} \)  // Advance write-set index
   12. function \( \text{READ}(\text{addr}_\text{e}) \)  // Read encoded value
      13. \( \text{addr}_\text{d} \leftarrow \text{DDECODE}(\text{addr}_\text{e}) \)
      14. if \( \text{fastMode} \) or \( \text{VALID}(*\text{addr}_\text{d}) \) then
         15. \( \text{return} *\text{addr}_\text{d} \)  // Return encoded mem value
      16. \( \text{val}_\text{e} \leftarrow \text{MRU}(\text{addr}_\text{e}, 0, \text{currIdx}) \)  // Find replica in log
      17. if \( \text{VALID}(\text{val}_\text{e}) \) then
         18. \( *\text{addr}_\text{d} \leftarrow \text{val}_\text{e} \)  // Fix memory with encoded value
         19. \( \text{return} \ \text{val}_\text{e} \)  // Return encoded value
      20. \( \text{ABORT} \)  // Abort because no valid state
   21. function \( \text{WRITE}(\text{addr}_\text{e}, \text{val}_\text{e}) \)  // Write encoded value
      22. if \( \neg \text{VALID}(*\text{addr}_\text{e}) \) then
         23. \( \text{ABORT} \)  // Abort because invalid operand
      24. \( \text{addr}_\text{d} \leftarrow \text{DDECODE}(\text{addr}_\text{e}) \)
      25. \( \text{writeSet}[\text{currIdx}] \leftarrow (\text{addr}_\text{e}, *\text{addr}_\text{d}, \text{val}_\text{e}) \)  // Append to write-set
      26. \( \text{currIdx} \leftarrow \text{currIdx} + 1 \)  // Update memory with encoded value
      27. \( *\text{addr}_\text{d} \leftarrow \text{val}_\text{e} \)  // Write encoded value
   28. function \( \text{ABORT} \)  // Abort a transaction
      29. if \( \neg \text{fastMode} \) then
         30. \( \text{KILL} \)  // Kill upon abort during self-healing
      31. \( \text{fastMode} \leftarrow \text{false} \)  // Enable self-healing
      32. \( \text{UNDO}((\text{writeSet}[\text{currIdx}], \text{startIdx})) \)  // Roll back state
      33. \( \text{currIdx} \leftarrow \text{startIdx} \)  // Reset write-set state
      34. \( \text{longjmp}(\text{ctxt}) \)  // Reset register state & jump to start
4 Transactional Encoding for Tolerating Transient Hardware Errors

<table>
<thead>
<tr>
<th>Corruption</th>
<th>Self-Healing Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬VALID(m.addr)</td>
<td>READ will ABORT because no read or MRU lookup is possible</td>
</tr>
<tr>
<td>¬VALID(m.val)</td>
<td>WRITE will ABORT because storing and logging is impossible</td>
</tr>
<tr>
<td>¬VALID(w.addr)</td>
<td>READ will ABORT because no MRU lookup is possible</td>
</tr>
<tr>
<td>¬VALID(w.val)</td>
<td>ABORT will KILL because no UNDO is possible without location</td>
</tr>
<tr>
<td>¬VALID(w.old)</td>
<td>WRITE will ABORT because the write-set would get corrupted</td>
</tr>
<tr>
<td>¬VALID(w.val) ∧ m.val ≠ w.val</td>
<td>READ will ABORT because indistinguishable which is latest version</td>
</tr>
<tr>
<td>Leads to crash</td>
<td>Catch signal and ABORT</td>
</tr>
</tbody>
</table>

Table 4.1: Overview of state and write-set corruptions how the error can be tolerated (m.*: memory state and w.*: current (most recently used) write-set entry).

Figure 4.4: Values in memory and write-set are masked using XOR to protect against exchanged operands and operators.

The READ is extended by a validation that checks if the value in memory is a valid code word before it is returned (see Algorithm 4.2, line 14). Together with failure atomicity, the validation protects from modified memory operands and faulty operations that produce invalid results and manifest in the state. If an invalid code word is encountered, we access the most recently used value from the replica (MRU, line 16). Note that the most recent entry for the requested address might not be in the boundaries of the current transaction but in the history of all previous transactions (see Figure 4.3). If the replicated value is valid, we correct the state by copying the value to memory and return it (lines 17–19). Otherwise, no valid state could be found and we must ABORT (line 20).

We additionally extend the encoding with an XOR mask for the detection of incorrect state and write-set accesses when an operation suffers from an exchanged operand or exchanged operator. If the address could be decoded successfully, it can still point to an incorrect memory location, e.g., caused by a bus error. Therefore, we XOR the current value with the encoded address and the index currIdx, as illustrated in Figure 4.4. The encoded address is also added to the values of the state in memory. It can detect if the encoded value belongs to the requested address (vm = ve ⊕ ae). The currIdx allows to check during DECODE if the correct entry is accessed by comparing the masked values in memory and at currIdx in the write-set (vm = vei ⊕ ae ⊕ i). The index i is determined by searching for the encoded address ae in the write-set. We then compare the masked value in memory at address ad with the partially unmasked value of the write-set entry (vm ⊕ i) for a match. This guarantees that memory...
values that are not stored at the correct memory address will be detected upon access. In any of the above cases, the XOR mask breaks the encoding of the value and can be detected during DECODE.

AN codes cannot detect if a valid but incorrect value is accessed, e.g., upon exchanged operands or lost updates. Accesses to incorrect values may lead to an inconsistent state. For our prototype, we rely upon signal handling to avoid crashes and recover when such an inconsistent state triggers exceptions and segmentation faults. This mechanism is used both in fast and self-healing mode. Recent work [DS12] has shown that transactions can be sandboxed transparently for POSIX C code. It additionally requires timeouts to break endless loops and recursion caused by errors.

4.6 EVALUATION

In this section, we evaluate and analyze the error tolerance and performance of our transactional encoding prototype (TE) that is implemented entirely in software. We are specifically interested in showing that (1) the majority of transient errors will indeed not result in a safety violation, (2) a fraction of errors not detected by encoded processing will be tolerated on a higher level using transactional memory (TM), and (3) the tolerance of errors introduces only a reasonable performance overhead compared to the error-free case.

We compare the different TE extensions against the native and encoded versions of the same application. All used combinations are summarized in Table 4.2.

We used the following test applications for the evaluation: bubble-sort (2500 numbers) and quick-sort (10^4 numbers) sort integers, crc32 computes a CRC-32 checksum (10^7 bytes), md5 computes the MD5 hash of a string (10^7 bytes), sha256 computes a SHA-256 checksum (10^7 bytes), and aes256 encrypts a buffer with AES in ECB mode (10^7 bytes). Our tests have been carried out on a machine with an Intel Core i7-3720QM CPU running Mac OS X 10.8. The applications were compiled using GCC 4.8 with inlining enabled.

4.6.1 TOLERANCE OF INJECTED ERRORS

To evaluate the error tolerance capabilities, we used the injection tool EIS, which is described in [Sch+10b]. EIS consists of a static and dynamic injector. The static injector instruments the module with trigger points capable of injecting one or multiple errors into each instruction at the level of LLVM bitcode according to the selected error symptom model. In our experiments we used the symptoms modified operands (MO), faulty operations (FO), exchanged operands (EOD), exchanged operators (EOT), lost stores (LS), as well as the aggregation of all types of transient faults (AT). For the error injection experiments we use small input parameters for the tested applications (bubble-sort: 10 numbers, quick-sort: 10 numbers, crc32: 10 bytes, md5: 10 bytes, sha256: 10 bytes, and aes256: 16 bytes), because they already result

<table>
<thead>
<tr>
<th>Variant</th>
<th>Encode</th>
<th>TM</th>
<th>Validate</th>
<th>Mask</th>
<th>exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoded</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TE_basic</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>TE_val</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>TE_mask</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TE_exc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.2: Combinations of the extensions.
Table 4.3: Ratio of tolerated transient errors (AT) using our approach with different extensions.

<table>
<thead>
<tr>
<th>Application</th>
<th>$\text{TE}_{\text{basic}}$</th>
<th>$\text{TE}_{\text{val}}$</th>
<th>$\text{TE}_{\text{mask}}$</th>
<th>$\text{TE}_{\text{exc}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble-sort</td>
<td>51.1 %</td>
<td>52.6 %</td>
<td>53.1 %</td>
<td>65.0 %</td>
</tr>
<tr>
<td>quick-sort</td>
<td>49.6 %</td>
<td>49.7 %</td>
<td>49.9 %</td>
<td>56.3 %</td>
</tr>
<tr>
<td>crc32</td>
<td>72.9 %</td>
<td>72.9 %</td>
<td>72.8 %</td>
<td>74.2 %</td>
</tr>
<tr>
<td>md5</td>
<td>22.5 %</td>
<td>59.7 %</td>
<td>59.5 %</td>
<td>67.6 %</td>
</tr>
<tr>
<td>sha256</td>
<td>12.9 %</td>
<td>76.9 %</td>
<td>76.8 %</td>
<td>79.6 %</td>
</tr>
<tr>
<td>aes256</td>
<td>56.0 %</td>
<td>57.7 %</td>
<td>58.5 %</td>
<td>71.1 %</td>
</tr>
</tbody>
</table>

Table 4.4: Comparison of the total size of the write-set and the number of unique addresses it contains.

<table>
<thead>
<tr>
<th>Application</th>
<th>Write-set size</th>
<th>Unique addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble-sort</td>
<td>6247500</td>
<td>2500</td>
</tr>
<tr>
<td>quick-sort</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>crc32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>md5</td>
<td>3125184</td>
<td>44</td>
</tr>
<tr>
<td>sha256</td>
<td>1250125</td>
<td>36</td>
</tr>
<tr>
<td>aes256</td>
<td>10001150</td>
<td>2501150</td>
</tr>
</tbody>
</table>

in a large number of trigger points. Note that this does not reduce the variety of instructions being covered. The dynamic injector runs the target application under supervision monitoring its behavior and checking its output against an error-free golden run. Each error symptom gets injected once at each possible trigger point in a single run. The applied modification is selected randomly for each trigger point, e.g., which bits get flipped. Depending on its outcome the runs are categorized as follows:

1. **Correct**: The application terminated successfully and produced the same output as the golden run.
2. **Detected**: The application aborted. This might be because (a) the operating system delivered a signal, or (b) the encoding detected an error that could not be corrected.
3. **Undetected**: The application terminated successfully but its output differs from the golden run, i.e., it silently corrupted data (SDC).
4. **Timeout**: The application exceeded its time limit (5 seconds) and was killed.

The outcome of our error injection experiments are presented in Figure 4.5. Overall, the figure shows that TE is able to tolerate a high degree of transient errors (AT). A large fraction (up to 79%) of detected or undetected native executions are converted into correct executions ($\text{TE}_{\text{exc}}$). Encoded processing already detects many injected error symptoms. A basic abort and retry of the erroneous transaction ($\text{TE}_{\text{basic}}$) already fixes many of those errors for bubble-sort, quick-sort, crc32, and aes256. The masking of $\text{TE}_{\text{mask}}$ proves to be ineffective. Its purpose is to protect against accesses of erroneously overwritten memory locations. However, if invalid addresses are detected while decoding or signaled by the operating system, the exception handling of $\text{TE}_{\text{exc}}$ can recover.

Table 4.3 contains a detailed list of the increase of correct executions when different extensions are enabled.

The encoded functions of md5 and sha256 do not return a result and, thus, no code word is validated at the end of the transaction. This manifests in a low tolerance rate for $\text{TE}_{\text{basic}}$. 

68
Figure 4.5: Behavior of the test applications when injecting errors according to the error symptom models.
because transactions are not aborted. Instead, the written incorrect state remains and is detected upon access in a following transaction, which will abort without returning to a correct state and enter self-healing mode. The self-healing mode uses the write-set to fix invalid loaded values, but cannot recover if the initial erroneous transaction already stored an invalid code word causing both the write-set entry and the memory to be invalid. The validation of each store of $TE_{val}$ is capable of correcting the state by rolling-back the erroneous transaction containing the erroneous computation.

### 4.6.2 PERFORMANCE OVERHEAD

We now want to evaluate the performance overhead introduced by TE in terms of time and space, executing in fast and self-healing mode. The overhead in time compared to the native execution is presented in Figure 4.6. Most of the slowdown is caused by encoded processing because it replaces all instructions to support arithmetic codes. The benefit of using commodity hardware is that it typically operates at higher clock-speeds. This amortizes the overhead compared to specialized hardware that has a longer time to market. In the error-free case TE operates in fast mode ($TE_{basic}$) and does not incur much additional overhead for most applications because it only appends new entries to the write-set. The inefficient bubble-sort algorithm represents the worst-case scenario as it swaps numbers unnecessarily often, hence yielding very large write-sets. $TE_{val}$ incurs the additional overhead to first validate the values to be stored. Enabling XOR masking results in a slightly higher overhead due to the computation of the mask.

The detection of an error aborts the transaction and enables the self-healing mode. The execution time gets extended by the roll back and re-execution with extensive validation. Figure 4.7 shows the ratio by which the execution is extended depending on the rate of detected errors. The overhead is the slowdown when comparing the error-free case in fast mode with the fault-tolerant execution in self-healing mode, modeled as:

$$Overhead_{avg} := Rate_{error} \times \frac{(Time_{abort} + Time_{heal})}{Time_{fast}}$$

For our applications, the overhead in execution time stays below 2.5% for an error rate up to 1%. Typically, the error rate of unreliable hardware is much lower.

The space overhead manifests in an increased memory consumption. Table 4.4 contains the properties of the write-set for all example applications. The size of the write-set represents the worst-case scenario for a roll back. An interesting property is the low number of unique
addresses in the write-set at the exit of most of the applications. It shows that the total size of the write-set can be reduced dramatically if garbage collection would be performed periodically.

### 4.7 SUMMARY

Depending on their safety integrity level, mission and safety critical systems typically require that a specified fraction of errors is masked, i.e., will not result in a safety violation. In this chapter, we have addressed the problem of how to ensure that a large fraction of execution errors will indeed be masked. Our solution combines encoded processing for detecting incorrect execution of code and guaranteeing data integrity, and transactional memory for recovering from errors using a checkpoint/rollback approach. The underlying idea is that, if one can detect and tolerate a sufficiently high fraction of failures using local retry, one can reach higher safety integrity level. Moreover, we investigated mechanisms that support repair of the state beyond what is possible by a simple checkpoint/rollback approach.

We have implemented a software prototype composed of an LLVM-based compiler for encoded processing and a lightweight transactional memory library optimized for dependability—rather than for concurrent synchronization. Experimental results show that a high degree of error tolerance can be achieved on unreliable hardware. The presence of errors has only a small impact on the overall performance, i.e., this mechanism could for example be used to lower the voltage of CPU cores to reduce their energy consumption.

![Figure 4.7: Cost of error tolerance for different error rates relative to the execution in fast mode.](image-url)
5 FASTLANE: IMPROVING PERFORMANCE OF SOFTWARE TRANSACTIONAL MEMORY FOR LOW THREAD COUNTS*

*The contents of this chapter first appeared at PPoPP ’13 [Warn+13a].
Software transactional memory (STM) can lead to scalable implementations of concurrent programs, as the relative performance of an application increases with the number of threads that support it. However, the absolute performance is typically impaired by the overheads of transaction management and instrumented accesses to shared memory. This often leads STM-based programs with low thread counts to perform worse than a sequential, non-instrumented version of the same application.

In this chapter, we propose FASTLANE, a new STM algorithm that bridges the performance gap between sequential execution and classical STM algorithms when running on few cores. FASTLANE seeks to reduce instrumentation costs and thus performance degradation in its target operation range, which makes the synchronization using STM more efficient. We introduce a novel algorithm that differentiates between two types of threads: One thread (the master) executes transactions pessimistically without ever aborting, thus with minimal instrumentation and management costs, while other threads (the helpers) can commit speculative transactions only when they do not conflict with the master. Helpers thus contribute to the application progress without impairing on the performance of the master.

We implement FASTLANE as an extension of a state-of-the-art STM runtime system and compiler. Multiple code paths are produced for execution on a single, few, and many cores. The runtime system selects the code path providing the best throughput, depending on the number of cores available on the target machine. Evaluation results indicate that our approach provides promising performance at low thread counts and for partitioned workloads executing on many cores: FASTLANE almost systematically wins over a classical STM in the 1-6 threads range, and often performs better than sequential execution of the non-instrumented version of the same application starting with 2 threads. We introduce a new benchmark that naturally supports the partitioning of data such that the benefit of reduced instrumentation costs scales with the number of partitions and allows FASTLANE to outperform classical STM for large numbers of threads.

5.1 INTRODUCTION

Transactional memory (TM) has received much attention over the last decade as it provides a scalable and easy-to-use approach to concurrent programming. Developers simply enclose critical sections within transactions1 that execute speculatively and abort when conflicting accesses to shared data are detected at runtime.

In a short term, software-based implementations of TM (STM) will remain in the focus for optimizations as processors with dedicated hardware TM (HTM) instructions are not commonly available yet or require a software fallback. Many applications make use of few threads only: They either run on a large fraction of computers that provide only a small number of processor cores (e.g., mobile devices), or they expose only little parallelism, which hardly exceeds a level of 3 threads for general purpose applications [Bla+10]. Therefore, our focus in this chapter is on STM for few threads. Note that systems with many cores also benefit from the efficiency improvements if they execute multiple processes with few threads each or if a workload is partitioned such that each partition is only accessed by a few threads at a time.

While STM implementations often exhibit excellent performance scalability with high thread counts [DSS06; DSS10; FC11; FFR08], the overheads related to transaction management and instrumentation of memory accesses2 are the main limitation when executing with few threads. In fact, the performance of a single-threaded non-instrumented application is generally higher than when using STM on a small number of cores [Cao+08; Chr+10; Dal+10;  

---

1TM-aware compilers typically provide higher-level atomic block language constructs that are transparently mapped to transactions.

2Reads and writes to shared memory are replaced by transactional accesses, which trigger execution of complex operations (conflict detection, maintenance of the read/write sets, etc.) requiring hundreds of additional cycles.
An in-depth study has shown that, with compiler instrumentation of transactions, on average more than four cores are necessary to outperform sequential code [Dra+11]. Let $x$ be the threshold on the number of cores necessary for STM to pay off. The goal of this work is to enable an application to outperform sequential code already with less than $x$ cores, i.e., to bridge the gap between single-threaded performance of the non-instrumented code and multi-threaded performance of the STM-based code on $x$ cores. To that end we propose FASTLANE, a novel synchronization strategy and runtime system designed to optimize performance of STM on $1 < n < x$ cores. Depending on the target architecture, the runtime system can select an optimal synchronization strategy for the application: sequential for 1 core, FASTLANE for 2 to $x$ cores, or STM for more than $x$ cores. Figure 5.1 depicts schematically the expected behavior of the three execution strategies and the zone where FASTLANE can boost performance as compared to the state-of-the-art STM algorithms.

The basic idea of FASTLANE is to have threads operate in one of two modes. One pessimistic master thread runs at nearly sequential speed with only minimal instrumentation, while all other threads execute speculatively and try to help the master whenever they can. The latter threads, called helpers, typically run slower than STM threads, because in addition to performing the extra bookkeeping associated with memory accesses they should not hamper progress of the master. The roles of master and helper can be changed dynamically by the runtime system during execution of the concurrent application, e.g., if a thread requests to perform irrevocable operations and must execute as master.

An application compiled for FASTLANE includes the different synchronization strategies (sequential, STM, and FASTLANE) to allow the selection of the appropriate strategy depending on the number of cores available on the target machine. For this, we have extended the DTMC compiler [Chr+10] so as to generate all the synchronization strategies within the application binary.

We have evaluated the performance of FASTLANE on a number of synthetic and realistic benchmarks, and compared them against STM and sequential executions. Our results show that FASTLANE performs competitively with sequential execution for a single thread, and performs most of the time better with already two threads. Further, FASTLANE often continues to scale well and generally outperforms STM algorithms up to six threads, which corresponds to the number of cores per processor on our test machine. When the workload can be partitioned, FASTLANE can execute one master thread per partition and performs significantly better than other STM algorithms even for high thread counts.

The rest of this chapter is organized as follows. Section 5.2 discusses related work. Sec-
tion 5.3 describes the FASTLANE algorithm and the design choices that led to several optimizations. Section 5.4 evaluates the performance of the algorithm on various synthetic and realistic benchmarks. Finally, Section 5.5 concludes.

5.2 RELATED WORK

A wide variety of efficient software transactional memory implementations have been proposed over the last few years [DSS06; DSS10; FC11; FFR08; HLR10; Sre+07]. The main focus has been on exploiting the available disjoint access parallelism with high thread counts. It has also been shown in previous work that dynamic tuning of the STM runtime system depending on the workload can significantly improve the throughput [RFF08; Spe10; Usu+09], e.g., the bookkeeping overhead can be reduced when no contention is present. Instead of tuning, FASTLANE focuses on optimizing the synchronization algorithm for few threads.

We are aware of only few STM designs that explicitly target small thread counts. Transactional mutex locks (TML) [Dal+10] use a versioned reader-writer lock: read-only transactions can concurrently execute and commit but, as soon as a transaction wants to write, it must acquire the lock, which will lead to an abort of all other active transactions. While no other transaction can execute concurrently when an update transaction is active, the benefit is that instrumentation overhead is minimal. Transactions only have to save the context upon start to support retries. No write or undo logs are needed, and transactional loads only have to check the status of the versioned reader-writer lock.

We compared FASTLANE to several optimistic STM implementations known for their efficiency and low instrumentation overhead. NOREC [DSS10] extends the idea of TML with buffered updates, a read-set and value-based validation to deal with concurrent updates. This allows read transactions to execute concurrently with an update transaction. Value-based validation requires a consistent state, i.e., the versioned lock must not be owned by an update transaction during the validation. If combined with FASTLANE, the master would hold the lock very frequently and for long periods, which hinders other threads to validate and would result in a poor performance. TINYSTM [FFR08] is a word-based implementation of the lazy snapshot algorithm [RFF06] that uses a shared array of revocable locks and time-based validation. The shared array allows a fine granular synchronization scheme that scales up to a high number of threads. It allows multiple update transactions to proceed in parallel if not in conflict. Therefore, FASTLANE uses such a shared array to allow the master and helpers to execute in parallel. TINYSTM can be configured in direct update mode (WT) or with buffered updates and encounter time locking (ETL).

Pessimistic transactions, like the master in FASTLANE, have been proposed in different flavors. A fully pessimistic STM [MS12] executes every transaction once and never aborts, enabling the execution of irrevocable operations and simpler debugging at the cost of limiting concurrent updates. ROBUSTM [Wam+10] (see Chapter 3) starts to execute transactions speculatively but gives transactions that aborted a certain number of times a priority privilege. This privilege lets a transaction pessimistically win all conflicts, even in the presence of crashes and non-terminating transactions, making it practically wait-free. Similarly, the authors of [Mal+10] propose to execute a transaction subject to a deadline in increasingly pessimistic modes as that deadline nears.

Runtime systems for parallelization often use speculation. Software lock elision [RHH09] processes critical sections speculatively in parallel but will fall back to lock acquisition upon frequent conflicts or irrevocable operations, which always wins over speculation. Other runtime systems auto-parallelize programs by thread-level speculation [OMH09] or profile-guided automatic loop parallelization [Meh+09]. Fastpath [Spe+09b] uses pessimistic and speculative modes to parallelize loops. Each iteration starts in speculative mode, using NOREC for synchronization, but can switch to a pessimistic mode that requires only entry and exit instru-
Table 5.1: Shared variables used by FASTLANE algorithms.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cntr</td>
<td>Counter that tracks updates of the master and helpers. The value is odd when a transaction performs updates and even otherwise. This variable is used for validation by the helpers.</td>
</tr>
<tr>
<td>dirty[]</td>
<td>Array of monotonically increasing integers. Each memory address is mapped to one entry in the array (by hashing the address modulo the size of the array). The entry contains the value of the counter cntr at the last time the address was written.</td>
</tr>
<tr>
<td>helpers</td>
<td>Lock to serialize commit attempts of helpers. It is implemented as a MCS list-based queue lock [MS91] and provides FIFO guarantees.</td>
</tr>
<tr>
<td>master</td>
<td>Lock to synchronize the master with the helper. Helpers must acquire helpers first. It is implemented as a TTAS (test-and-test-and-set) lock [KRS88] and protects all shared variables.</td>
</tr>
<tr>
<td>masterID</td>
<td>Identity of the current master thread. It must only be modified after the master has been acquired.</td>
</tr>
</tbody>
</table>

5.3 FASTLANE ALGORITHMS

The high-level objective of FASTLANE is to perform (1) approximately identically to sequential execution, and (2) better when leveraging a few additional threads. To meet the first goal, we rely on a pessimistic and lightly-instrumented master thread that never aborts and, hence, should provide performance similar to sequential execution on a single core. The role of the helper threads is to address the second objective, i.e., improve performance by committing transactions that do not conflict with those of the master.

We start by describing the global data structures used by FASTLANE and the behavior of the master thread, before describing the helper threads and the optimizations we applied.

5.3.1 DATA STRUCTURES

The shared data structures used by the FASTLANE algorithms are summarized in Table 5.1 and illustrated in Figure 5.2. They essentially consist of: a shared counter, cntr, that is incremented each time a transaction commits updates; a shared array of integers, dirty[], that protects a set of memory addresses and stores the value of the counter at the last time one of these addresses was updated; a FIFO lock, helpers, implemented using the scalable MCS\(^3\) algorithm [MS91], to serialize helpers that intend to commit update transactions; a lock, master, implemented using the low-overhead TTAS\(^4\) protocol [KRS88], which must be acquired before any shared data can be modified; and a variable, masterID, that holds the current master.

---

\(^3\)An MCS lock uses a list-based queue to grant access to the lock in FIFO order. Threads only spin on cacheable local memory, thus achieving high scalability.

\(^4\)A TTAS (test-and-test-and-set) lock scales better than a simpler test-and-set spin lock. TTAS first reads the state of the lock using a normal memory access and, if it is free, tries to acquire it using an atomic operation. This helps avoid unnecessary traffic on the bus because the requesting thread will spin on its local cache if the lock is not free. A TTAS lock can be released by a simple write to memory.
5.3.2 CODE PATH SELECTION

We have extended the DTMC open-source C/C++ TM compiler [Chr+10] to generate multiple code paths for each transaction (see Figure 5.3): (1) a sequential path without instrumentation of reads and writes, (2) a pessimistic master path with a lightweight instrumentation of writes, (3) a speculative helper path with instrumentation for reads and deferred writes, and (4) an optimistic fully-fledged STM path with instrumentation of reads and writes.

The selection of the code path is performed dynamically at the beginning of a transaction. As long as only a single thread is registered with the runtime system, the sequential code path will be executed. The FASTLANE mode with the master and helper code paths is enabled after a second thread is active. When more than a specified number of threads are registered, the STM mode is selected. This number is currently fixed empirically based on observations of the average number of threads after which STM outperforms FASTLANE. Switching to STM mode requires the acquisition of a quiescence lock that prevents further helper transactions from starting, as well as the acquisition of the master lock to stop the master. When no threads execute transactions, it is safe to set the code path for all subsequent transactions to the STM algorithm.

Algorithm 5.1 shows the selection of the master and helper in the FASTLANE mode. Initially, the thread that registered first with the runtime system becomes the master. When the master calls START to begin a transaction (lines 1–2) it must first acquire master using TTAS to synchronize with the helpers (line 3). As the master transaction will apply updates in-place during its execution, master must be owned during that time. If another thread was meanwhile promoted to master, the thread releases master and continues as helper (lines 7–10, Section 5.3.4). For the common case that the master remains unchanged, one can assume that masterID is cached, hence, the overhead of the additional check is negligible. The thread can continue as master (lines 4–6, Section 5.3.3). Note that transactions that have an explicit abort request (cancel) cannot be executed as master. The compiler identifies such transactions and generates code that always selects the helper code path.

Helper threads will acquire master during their HELPER COMMIT because they execute speculatively in parallel with the pessimistic master. They directly jump to the helper code path (lines 15–17, Section 5.3.4).

Helper threads can also request to gain master privilege, e.g., to perform irrevocable operations such as I/O or system calls (see Section 5.3.5). Upon such an event, the thread aborts and sets the pessimistic flag when the transaction is restarted (line 11). A thread that wants to become master first acquires master using TTAS (line 12). It can now execute in isolation and set masterID to its thread identifier to reflect the change (line 13). It then continues with the master code path (line 14).

Figure 5.2: Illustrations of shared variables used by FASTLANE algorithms.
Figure 5.3: Multiple code paths are generated for each transaction and can be selected at transaction start by the runtime system.

Algorithm 5.1: Code path selection at transaction start.

```
function START(pessimistic)
    // Begin or restart a transaction
    if masterID = threadID then
        // If thread is master
        ttas-lock(master) // Acquire master
        if masterID = threadID then
            // If thread still master
            MASTERSTART // Start transaction as master
            return CP_MASTER // Jump to master code path
        else
            // Thread is helper
            ttas-unlock(master) // Release master
            HELPERSTART // Start transaction as helper
            return CP_HELPER // Jump to helper code path
    else if pessimistic then
        // If master privilege requested
        ttas-lock(master) // Acquire master
        masterID ← threadID // Set current thread as master
        return CP_MASTER // Jump to master code path
    else
        // Thread is helper
        HELPERSTART // Start transaction as helper
        return CP_HELPER // Jump to helper code path
```

5.3.3 MASTER THREAD

The operation of the master thread is described in Algorithm 5.2. After the selection of the master code path in Section 5.3.2 the thread already owns \texttt{master} at MASTERSTART (line 2) and has exclusive privilege to directly update shared data during the pessimistic transaction.

Algorithm 5.2: Master code path.

```plaintext
1 function MASTERSTART
   // Thread owns \texttt{master} at this point, no context saved
2 function MASTERRead(addr)
   return *addr // No instrumentation
3 function MASTERWrite(addr, val)
   if ¬(cntr & 0x01) then // Is \texttt{cntr} already odd?
      cntr ← cntr + 1 // Master sets odd \texttt{cntr} once
   dirty[hash(addr)] ← cntr // Mark as modified
   *addr ← val // No additional bookkeeping
4 function MASTERCOMMIT
   if cntr & 0x01 then // Is \texttt{cntr} odd from write?
      cntr ← cntr + 1 // Master sets \texttt{cntr} even
   ttas-unlock(master) // Release \texttt{master}
```

Instrumentation of memory accesses is minimal on the master thread so as to obtain performance as close as to a single thread case. MASTERRead operations are not instrumented (line 4) because the master does not need to ever validate the read set, while MASTERWrite operations are augmented by a store of the value of the counter in the corresponding entry of the dirty[] array (line 8). No undo logging is required because the pessimistic master never aborts. On the first write only, \texttt{cntr} must be incremented to an odd value (line 7). Finally, upon MASTERCOMMIT, the master simply reverts the counter to an even value in case it has performed writes (line 12) and releases \texttt{master} (line 13).

In most cases, the master has very low overhead. The \texttt{masterID} variable is infrequently modified and thus remains in the CPU cache. At transaction START, only one atomic test-and-set operation is needed to implement the TTAS lock. Between the TTAS lock and unlock operations, only the master can write shared data and thus we do not need any additional atomic operations or barriers. MASTERRead operations are not instrumented and MASTERWrite operations go directly to memory. Upon the first write, \texttt{cntr} is incremented using a simple store, which may cause invalidation messages if \texttt{cntr} is cached in other cores. Note that \texttt{cntr} cannot be updated by other threads while \texttt{master} is owned, thus subsequent MASTERWrite operations have lower overhead because \texttt{cntr} is cached and unmodified. If the transaction is read-only we completely avoid the invalidation of \texttt{cntr}. Finally, one update to the dirty[] array is necessary for every write. Upon MASTERCOMMIT, the \texttt{master} lock is released in order to serialize all changes made by the transaction. In Section 5.3.6 we show how the contention on \texttt{master} and \texttt{cntr} can be reduced.

5.3.4 HELPER THREAD

The price to pay for having a lightly instrumented master thread becomes clear when considering the algorithm of the helpers. Extra work must be performed to speculatively execute transactions and try to commit changes without slowing down the master.

The functions of the FASTLANE helper code path are shown in Algorithm 5.3. Upon HELPER-START, the current value of \texttt{cntr} is stored for subsequent validation purposes, discarding the
Algorithm 5.3: Helper code path

1 function HELPERSTART
2 setjmp(ctxt) // Store the current context
3 start ← cntr & ~1 // Take even counter value

4 function HELPERREAD(addr)
5 if CONTAINS(write-set, addr) then // Already written?
6 return GET(write-set, addr) // Return written value
7 val ← *addr // Read from memory
8 if dirty[\text{hash}(addr)] > start then // Validate read
9 ABORT // Abort and restart
10 ADD(read-set, addr) // Add address to read set
11 return val

12 function HELPERWRITE(addr, val)
13 if dirty[\text{hash}(addr)] > start then // Validate write
14 ABORT // Abort and restart
15 PUT(write-set, addr, val) // Add to (or update) write set

16 function VALIDATE // Validate read and write sets
17 if cntr ≤ start then // No updates since start
18 return true
19 foreach addr ∈ (read-set ∪ write-set) do
20 if dirty[\text{hash}(addr)] > start then // Concurrent update
21 return false
22 return true

23 function HELPERCOMMIT
24 if EMPTY(write-set) then // Read-only transaction?
25 return // Commit immediately
26 mcs-lock(helpers) // Acquire helpers
27 ttas-lock(master) // Acquire master
28 if ¬VALIDATE then
29 ttas-unlock(master) // Release master
30 mcs-unlock(helpers) // Release helpers
31 ABORT // Abort and restart
32 cntr ← cntr + 1 // Helper sets cntr odd
33 foreach (addr, val) ∈ write-set do
34 dirty[\text{hash}(addr)] ← cntr // Update dirty[]
35 *addr ← val // Update memory
36 cntr ← cntr + 1 // Helper sets cntr even
37 ttas-unlock(master) // Release master
38 mcs-unlock(helpers) // Release helpers

39 function ABORT
40 CLEAR(read-set, write-set) // Clear read and write-set
41 longjmp(ctx) // Jump back to start
least significant bit to force the value to be even (line 3).

For HELPERREAD operations, the helper first checks whether it has already written to the same address. If so, it returns the value of the previous write (lines 5–6). Otherwise, it reads the value and conservatively checks if the address has been concurrently written, by validating the associated entry of dirty[]. If so, the transaction simply aborts (lines 7–9). This guarantees opacity [GK08]. Otherwise, the read can successfully complete: the address is added to the read set and the previously read value is returned (lines 10–11).

Upon HELPERWRITE, we check if the written address has possibly been updated concurrently, like for reads, and if so, the transaction aborts (lines 13–14). Otherwise, we simply add or update the address and the written value in the write set (line 15), delaying the actual update of the shared memory to the commit phase.

The HELPERREAD and HELPERWRITE operations must both perform lookups on the write set for each invocation. The write set is a vector with an index to reduce the lookup time, following the same general principle as in [Spe+09a].

The main idea of HELPERCOMMIT is to perform the validation of the read and write sets, resulting in either an abort or a successful commit, while holding master. If the transaction is read-only (lines 24–25), all memory accesses have already been validated by the HELPERREAD operation and the transaction can commit immediately. Otherwise, the helper must ensure mutual exclusion for its commit phase. To that end, it first acquires the MCS queue lock helpers (line 26) to synchronize with other helpers and then acquires master using TTAS (line 27) to synchronize with the master. The rationale behind using the additional helpers lock is to reduce the contention on master, i.e., minimize the negative impact of the helpers on the master. In Section 5.3.6 we use the helpers for a lock handover optimization.

The validation is performed while holding master and no other thread (even the master) can interfere. VALIDATE verifies if any address stored in the read and write set may have been concurrently updated, by looking into the dirty[] array (lines 16–22), and if so HELPERCOMMIT conservatively aborts after releasing helpers and master (lines 28–31). Upon successful validation, all pending updates stored in the write set are sent to shared memory (line 35) and the associated entries of the dirty[] array are updated with the current odd cntr (lines 32 and 34). Finally, cntr is increased to the next even value (line 36) and master and helpers are released (lines 37–38).

5.3.5 IRREVOCABILITY AND PRIVATIZATION

Some operations cannot be executed speculatively because they cannot be reverted, e.g., system calls or other operations with externally visible side effects. If a thread encounters such an operation and is currently executing speculatively, it requests to enter the pessimistic mode to ensure that the transaction must not abort. Depending on the current code path, this requires for a helper thread to become the master and for an STM thread to execute exclusively in sequential mode. Note that switching to the pessimistic modes can also be used when the progress of a thread is at stake, with the benefit of the master to allow parallel helpers in contrast to the exclusive sequential mode.

A thread in STM mode that requests irrevocability must acquire a quiescence lock, a common approach described in detail in the literature [Wan+07]. In short, it will prevent other threads from starting transactions. It then waits until all active transactions are either committed or aborted. After that it can execute the transaction in isolation, using a non-instrumented code path, and release the quiescence lock after commit.

In FASTLANE mode, the pessimistic master is used to execute irrevocable operations that cannot be rolled back. Helpers must abort the transaction when they encounter irrevocable operations, switch to the master code path (presented in Section 5.3.2) to re-execute the transaction. Other helpers can continue to execute transactions speculatively in parallel. Note
that the operations of the master must not contain non-transactional code that does not reflect updates to shared memory in dirty[].

Threads can request to privatize data in order to access it outside of transactions afterwards. Privatization safety [HLR10] requires that no other threads are accessing the data after the privatizing transaction committed. This is supported natively for the master because, once it has started its privatizing transaction, no helper can commit concurrently before it is finished. The helpers will abort when their validation encounters data privatized by the master. Helper threads that request privatization must wait after they committed until all earlier helper transactions are either committed or aborted. This is determined by looping through all transaction descriptors and waiting if they are active and still have a start counter value less than the privatizing transaction.

5.3.6 OPTIMIZATIONS

The goal of FASTLANE is to reduce the overhead for master transactions in order to achieve a performance close to sequential non-instrumented execution. The lightweight instrumentation of the master presented in Section 5.3.3 combined with a distinct compiler generated code path is key to a range of optimizations: all master-specific instrumentation can be inlined to remove call overheads; because the master only updates global data, it does not need access to a transaction descriptor during its read or write operations; and at transaction start, the master can omit saving the context because it never aborts.

Helper threads that execute in parallel will have an impact on the master because master, cntr and dirty[] are shared resources. While the contention on dirty[] is spread over the elements of the array, the contention on master and cntr can have a negative impact because they will be modified by each update transaction. With high likelihood, cntr will be in the cache of the helper threads for validation purposes, thus the modifications will cause invalidation traffic.

We applied a number of specific optimizations to the FASTLANE algorithm to further reduce the overheads of the master and helper threads:

1. **Keep-lock** — The master thread can keep the master lock if no helper requests it to avoid unnecessary updates to shared variables.
2. **Pre-validate** — Helper threads can validate before attempting to acquire the master lock, to stop the master only if they have a high likelihood of a successful commit.
3. **Hand-over** — Helper threads can hand over the master lock if a successor exists in the helpers MCS queue to increase the commit chance.

The first optimization (keep-lock) can save the master the cost of releasing and reacquiring master, along with incrementing cntr, as long as no helper requests it. After each MASTERCOMMIT, the master checks if helpers was acquired. If this is not the case, the master keeps master and does not increment cntr to the next even value (line 3 in Algorithm 5.1 and lines 11–13 in Algorithm 5.2 will not be executed). A helper acquires helpers when it needs to commit and must acquire cntr, or when the helper aborts because it cannot validate. In the latter case, the helper needs the counter to be incremented to eventually commit. Indeed, assume that a shared variable has been written by the master when the value of the counter is x (odd) without the counter being subsequently incremented. A helper that later reads the same variable will remember x−1 (even) as start value of the counter and will systematically fail validation until cntr becomes greater than x (see Algorithm 5.3, lines 3 and 20).

Helper threads can reduce the contention on cntr by validating before they attempt to acquire master with the second optimization (pre-validation). After acquiring helpers (Algorithm 5.3, line 26) the thread calls VALIDATE and keeps the current value of cntr. Upon

⁵It is sufficient to check if the head node of the helpers MCS queue is not null.
successful validation it acquires `master` (line 27) and must only validate again (line 28) if `cntr` was incremented because another thread committed in the meantime. This is expected to reduce contention with the master, as a transaction that is known to abort will not compete for `master`. We prevent multiple helpers from committing concurrently using the `helpers` lock in order to avoid interference with the pre-validation from other helpers.

Finally, the third optimization (hand-over) allows the helper threads to hand over the `master` lock if a successor is waiting in the `helpers` MCS queue at `HELPERCOMMIT`. In that case, the helper does not increment `cntr` to the next even value and skips releasing the `master` lock (Algorithm 5.3, lines 36–37). It only releases `helpers` (line 38) and the succeeding owner in the queue of `helpers` can skip the acquisition of `master` (line 27) and will increment `cntr` by two to the next odd value (instead of line 32). Besides a reduction of the contention on `master`, this optimization is expected to improve the chance for helper threads that have already reached the commit phase to complete it. This is particularly interesting when combined with the second optimization: the pre-validation holds because no other thread can commit in the meantime.

### 5.4 EVALUATION

In this section, we evaluate and analyze the performance of FASTLANE. We compare the FASTLANE algorithm against non-instrumented sequential execution; two STM variants that are based on the lazy snapshot algorithm [RFF06]: TINYSTM [FFR08] operating either in write-through mode (WT), i.e., direct updates to memory, or in write-back mode with encounter time locking (ETL), i.e., buffered updates with eager conflict detection; and two STMs based on a single versioned lock, either exclusively directly updating memory (TML [Dal+10]) or performing buffered updates and value-based validation (NOREC [DSS10]). For FASTLANE, we measured the plain algorithm (FL, see Sections 5.3.3 and 5.3.4) as well as the optimizations described in Section 5.3.6. All evaluated configurations are summarized in Table 5.2.

For our test applications, we use the synthetic `intset` micro-benchmarks, realistic applications from the STAMP [Cao+08] benchmark suite, and a new benchmark that we designed to analyse the benefits and limitations of FASTLANE and which computes communities of interest for communication networks (see Section 5.4.5).

The `intset` benchmarks perform randomly queries and updates on integer sets implemented as a `red-black tree` (RB), a `linked list` (LL), a `skip list` (SL), or a `hash set` (HS). We use a working set of 8,192 elements for RB; 2,048 and 1,024 elements for LL; and 1,024 elements for SL and HS. We use update-to-lookup ratios of 5% and 20%, and the execution time for each run is 10 seconds.

The STAMP benchmark suite consists of the following applications: `bayes` learns the structure of Bayesian networks in a directed acyclic graph; `genome` performs gene sequencing using hash sets and string search; `intruder` emulates a signature-based network intrusion detection system by matching packets against signatures stored in self-balancing trees; `labyrinth` finds the shortest-distance paths between pairs of points using breadth-first search; `kmeans` clusters a set of partitioned points in parallel; `ssca2` constructs an efficient graph data structure using adjacency arrays; `vacation` emulates a travel reservation system, reading and writing different tables that are implemented as red-black trees; finally, `yada` performs mesh refinement of triangles in a work queue.

Using FASTLANE, there is a very unbalanced workload distribution between threads because the master is able to process transactions much faster than the helpers. Therefore we have adapted the STAMP benchmarks with a partitioning-based dynamic work balancing that introduces only very little overhead and allows adapting the amount of work for each thread and account for differences in throughput between the master and helpers. Otherwise, all STAMP benchmarks are configured accordingly to the documentation with parameters for
5 FASTLANE: Improving Performance of STM for Low Thread Counts

Our main goal is to achieve better scalability for low thread counts than traditional STM approaches. Figure 5.4 presents the throughput obtained in millions of transactions per second for the duration of each of the intset benchmarks. Figure 5.5 presents completion times for STAMP applications, which have a fixed number of transactions depending on the input parameters. The STAMP graphs show execution times instead of scalability to allow an easy comparison with the sequential baseline Overall, FASTLANE scales well for up to 6 threads and is on average more efficient than STM approaches for that number of threads. The sequential baseline corresponds to an implementation of transactions using a global lock because all

non-simulator runs and high contention.

Our tests have been carried out on a dual-socket server with two 6-core Intel Xeon Westmere-EP X5650 running 64-bit Linux 3.0. All 6 cores of a processor share the L3 cache. The CPU affinity was configured such that the penalty of moving data between sockets is as limited as possible, i.e., for up to 6 threads only a single processor is used. All benchmarks were compiled with the DTMC open-source TM C/C++ compiler [Chr+10].

In the rest of this section, we first analyze the scalability of FASTLANE against existing STM algorithms for a low number of threads. Then, we study the comparative contribution of the master and the helper threads. We later study the impact of optimizations. Finally, we present the new benchmark that computes communities of interest for communication networks.

5.4.1 SCALABILITY FOR LOW THREAD COUNTS

Figure 5.4: Throughput of the Intset benchmarks (higher is better).
Table 5.2: STMs and FASTLANE configurations used in the tests.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Non-instrumented sequential execution.</td>
</tr>
<tr>
<td>WT</td>
<td>TINYSTM operating in write-through mode (direct updates to memory) [FFR08].</td>
</tr>
<tr>
<td>ETL</td>
<td>TINYSTM operating in write-back mode with encounter time locking (buffered updates with eager conflict detection) [FFR08].</td>
</tr>
<tr>
<td>TML</td>
<td>Single versioned lock with exclusive direct updates to memory [Dal+10].</td>
</tr>
<tr>
<td>NOREC</td>
<td>Single versioned lock with buffered updates and value-based validation [DSS10].</td>
</tr>
<tr>
<td>FL</td>
<td>Basic version of FASTLANE with no optimizations (Sections 5.3.3 and 5.3.4).</td>
</tr>
<tr>
<td>FL-PV</td>
<td>FASTLANE with pre-validation before the helpers stop the master (Section 5.3.6).</td>
</tr>
<tr>
<td>FL-HO</td>
<td>FASTLANE with hand-over of locks for helper threads and keep-lock for master threads (Section 5.3.6).</td>
</tr>
<tr>
<td>FL-O3</td>
<td>FASTLANE with all three optimizations enabled: keep-lock, pre-validation, and hand-over (Section 5.3.6).</td>
</tr>
<tr>
<td>FL-P</td>
<td>FASTLANE with support for partitions (one master per partition, see Section 5.4.5).</td>
</tr>
</tbody>
</table>

Figure 5.5: Completion times of the STAMP benchmarks [Cao+08] (all with high contention configuration, lower is better).
benchmarks spend most of their time inside transactions and achieve no scalability.

On the intset micro-benchmarks, the minimal overhead of the master thread gives it a head start and the helpers contribute their share when the number of threads increases. FASTLANE is only outperformed on the RB micro-benchmark by TML, because of the instrumentation overhead to access dirty[] in FASTLANE. TML is typically very efficient for workloads that contain mostly transactions that are either short, read-only, or have a short period of updates at the end, but cannot exploit parallelism as long as a single update transaction is active.

The intset micro-benchmarks have a drop in throughput when more than 6 threads are active and the second socket is in use. This due to the more expensive cache coherence traffic over the interconnect, for up to 6 threads the L3 cache can handle the coherence. Here, the main source of cache contention, common to all STMs and FASTLANE, is the shared clock used for versioning that must be incremented upon each update transaction.

The STAMP benchmarks do not exhibit this behavior because they have larger transactions and, hence, the relative impact of the cache contention bottleneck is lower. Here, FASTLANE wins over all STMs for up to 6 threads with the exception of labyrinth, intruder and ssca2 for which ETL is more efficient. With the two latter benchmarks, the master performs short update transactions at a high rate that prevent the helpers from committing.

Note that the performance of some algorithms is not shown for bayes and labyrinth because the DTMC compiler instruments loading of regions, which is currently not supported by NOREC and results in prohibitively long execution times for WT. Hence, both variants are omitted from the graphs. The other algorithms serialize transactions upon such an operation: FASTLANE switches to master mode; ETL acquires the quiescence lock and executes the non-instrumented sequential code path; and TML acquires its reader-writer lock. As a result, labyrinth does not exhibit scalability and the plot only shows the constant instrumentation overhead for the FASTLANE master and TML, while ETL executes non-instrumented code. Bayes scales beyond sequential execution because not all time is spent inside serialized transactions, leaving a portion of the application where parallelism can be exploited.

Even if FASTLANE shows better performance than STMs on yada, it does not scale. The reason is that yada spends most of its time in long-running transactions and that FASTLANE serializes transactions of the master thread and commits of helper transactions by the master lock. As long as the master executes a transaction, no helper can commit and when a helper wants to commit it must stop the master.

5.4.2 CONTRIBUTION OF THE MASTER

To better understand the performance of FASTLANE, we first evaluate the overhead of the master thread with respect to the non-instrumented sequential execution. Since we run the plain FASTLANE algorithm without dynamic switching of code paths, using one thread amounts to using only the master thread. In that case, instrumentation is lightweight: it only needs to acquire and release master upon beginning and committing a transaction, respectively. Loads have no instrumentation at all, while writes only require an additional update to the dirty[] array and the first write additionally increments cntr.

The one thread results in Figure 5.4 and Figure 5.5 show that the master can indeed achieve single-threaded throughput close to that of sequential execution. For intset, the performance is very close to sequential: less than 2% slower for LL, at most 34% for RB, and 16% on average. For STAMP, the overhead ranges from 5% for kmeans to 52% for yada, with an average of 29%.

This good performance can be explained because master and cntr are cached and have only a marginal impact on the overhead. With an increasing update rate the overhead slightly increases because dirty[] must be updated more often. Since the optimization that keeps master does not perform noticeably faster than the basic algorithm, the updates to dirty[] are the main source of overhead for the master.
When comparing FASTLANE to the state-of-the-art STM algorithms, the latter require non-trivial algorithms to be executed for every transactional operation. While runtime systems could decide to choose the sequential non-instrumented path if only a single thread is active, we are interested in the general overhead introduced by an algorithm during its normal operation, i.e., the instrumented code path but without contention. STMs must typically copy the current CPU context at transaction START to support restart upon abort, keep track of read and write sets upon memory accesses, and perform validation and memory copy operations upon commit. FASTLANE objective is to streamline these costs for the master thread.

TINYSTM and NOREC suffer from high transactional management costs that are mainly depending on the number of transactional memory accesses, e.g., LL has the largest transaction sizes and the biggest performance degradation. TML has slightly higher overhead than FASTLANE because it must additionally save the context at START and instrument reads to check for concurrent writers. These observations are also visible in the STAMP measurements. Only bayes and labyrinth differ, because they contain compiler instrumented regions that will be executed in irrevocable mode by ETL (non-instrumented) and by the master with constant overhead in FASTLANE.

### 5.4.3 CONTRIBUTION OF THE HELPERS

To understand the importance of helpers in the global commit throughput of applications, we show in Table 5.3 the percentage of commits that have been achieved by helpers for all FASTLANE variants and all applications, with 2 and 6 threads. For the intset micro-benchmarks the contribution of the helpers varies from approximately 15-40% with 2 threads, to 60-85% with 6 threads. There is no noticeable difference between the FASTLANE variants. This can be explained by the fact that transactions are short and all identical, which limits the benefits of pre-validation and lock hand-over.

For the STAMP benchmark, one can observe important differences between the applications and the FASTLANE variants. Helpers contribute almost nothing with the bayes, intruder, and
Table 5.4: Impact of optimizations for FL-O3: percentage of failed pre-validations; percentage of pre-validations that hold after acquisition of the master lock; and hand-overs of helpers.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Failed PV (%)</th>
<th>PV holds (%)</th>
<th>HO used (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 thr.</td>
<td>6 thr.</td>
<td>12 thr.</td>
</tr>
<tr>
<td>rb 8k 05%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>rb 8k 20%</td>
<td>0.0</td>
<td>0.1</td>
<td>1.4</td>
</tr>
<tr>
<td>ll 1k 05%</td>
<td>0.0</td>
<td>0.1</td>
<td>2.9</td>
</tr>
<tr>
<td>ll 2k 05%</td>
<td>0.0</td>
<td>0.1</td>
<td>1.3</td>
</tr>
<tr>
<td>sl 1k 05%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>sl 1k 20%</td>
<td>0.0</td>
<td>0.6</td>
<td>4.0</td>
</tr>
<tr>
<td>hs 1k 05%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>hs 1k 20%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>bayes</td>
<td>0.0</td>
<td>45.3</td>
<td>68.6</td>
</tr>
<tr>
<td>genome</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>intruder</td>
<td>0.0</td>
<td>54.9</td>
<td>86.8</td>
</tr>
<tr>
<td>labyrinth</td>
<td>0.0</td>
<td>6.2</td>
<td>53.1</td>
</tr>
<tr>
<td>kmeans</td>
<td>0.0</td>
<td>50.6</td>
<td>72.8</td>
</tr>
<tr>
<td>sscas2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>vacation</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>yada</td>
<td>0.0</td>
<td>0.0</td>
<td>89.7</td>
</tr>
</tbody>
</table>

yada applications because of the size of transactions that leave almost no opportunity for helpers to commit without conflicts. One should also point out that bayes and labyrinth have very few transactions (less than 2,000 for the whole execution) and, hence, results are not very representative.

With genome, sscas2, and vacation, we observe the benefits of the hand-over optimization. When executing with 6 threads, the contribution of helpers in genome increases from 6.3% to 50.9%, i.e., almost one order of magnitude. Gains are also significant with the other two applications.

Interestingly, kmeans appears to suffer from the hand-over optimization but benefits from pre-validation. Indeed, with 6 threads, the contribution of the helpers increases from 1.4% to 29% when activating the latter optimization. This is because pre-validation helps detect early that a transaction is doomed and must abort, without needing to acquire the locks and slow down other threads. This optimization is particularly important given the level of contention in kmeans.

### 5.4.4 IMPACT OF OPTIMIZATIONS

We now focus on the the impact of optimizations. Table 5.4 shows, for FL-O3, the following metrics for 2, 6, and 12 threads: (1) percentage of failed pre-validations, i.e., helper transactions abort before even trying to acquire the master lock and hence do not slow down the master; (2) percentage of pre-validations that still hold after acquisition of the master lock, i.e., helper transactions do not need to validate again; and (3) percentage of committed helper transactions that benefited from the hand-over optimization.

One can first notice that pre-validation almost never fails for the intset benchmarks and several of the STAMP benchmarks. It only fails at a significant rate with bayes, intruder, and kmeans, as well as with labyrinth and yada but only when many threads are used. These numbers are consistent with the nature of the workloads: pre-validation is more likely to fail
5.4 Evaluation

Figure 5.6: Windows are merged into the communities of interest. Each caller has an ordered Top-K list of callees.

when conflicts with concurrent transactions are frequent. This optimization saves the cost of acquiring the master lock and slowing down the other threads.

For most benchmarks, pre-validation does not hold often. This is because helper threads typically request the master lock while the master thread is active. Once they succeed in acquiring it, the master has committed and incremented \( cntr \), which requires helpers to validate again. Only for some STAMP applications (bayes, labyrinth, and kmeans) does pre-validation hold more than 10% of the time, and essentially in the case of 2 threads. These benchmarks have long-running transactions that reduce the likelihood of a concurrent commit.

Finally, we observe that the hand-over optimization is very effective in STAMP benchmarks that have workloads with transactions of different sizes, but less so for the intset benchmarks because transactions are short and identical, hence the master commits at a high rate and prevents helpers handling over the lock to one another. As expected, hand-overs are generally more frequent when increasing the number of threads.

5.4.5 COMMUNITIES OF INTEREST BENCHMARK

In many applications, data can be naturally partitioned such that transactions operating on different partitions do not conflict and threads rarely access the same partition at the same time. Such applications can take full advantage of the FASTLANE algorithm because each partition can conceptually have its own master. If several threads try to access a partition at the same time, all but the first one will do it speculatively as a helper.

To evaluate the benefits of FASTLANE in such settings, we developed a new benchmark implementing an operator from the field of streaming and batch systems that calculates the communities of interest (COI) for communication networks [CPV01; WHF11]. The COI benchmark [WW13] implements an operator that processes telephone calls and calculates the callees that are most often called by a caller. Figure 5.6 schematically illustrates the operator. One transaction will merge a given sequence of calls (“window”) into the COI. The window has a configurable size \( W \) and is generated randomly before each transaction. The identities of callers and callees are randomly distributed between 1 and \( N \), and the duration of each call between 1 and 60 seconds.

The COI is computed for each caller that placed a call. The operator maintains a top-k list of size \( K = 9 \) in which callees are ordered by weight. The weight is calculated as a moving average with factor \( \theta \) (\( 0 \leq \theta \leq 1 \)): \( \text{weight of callee in top-k} \times \theta + \text{duration in window} \times (1 - \theta) \). Thus, \( \theta \) defines how much a new record influences the data. As the transaction iterates over the window, it updates the weights of callees (possibly inserting new entries) and moves
them in the right position in the top-k list.

The partitioning of the data is based on the identity of the caller in the COI. We extended FASTLANE to support partitions in the following way: (1) each partition has its own master, cntr, and helpers; (2) the hash function to find the dirty[] entry was adapted to be aware of the partition and keeps disjoint partitions in dirty[] during the mapping; (3) at START each thread becomes master if there is no other master, and releases the master privilege on MASTERCOMMIT; (4) all optimizations are disabled. Note that the approach taken in this benchmark differs from related work on TM partitioning, e.g., [RFF08], because we do not rely on tuning to determine which synchronization mechanisms to use in each partition, and we can benefit from more parallelism than a shared lock (a secondary thread can access the partition as a helper) and less overhead than multiple locks as used by many STMs.

Figure 5.7 shows the experimental results for FASTLANE with partition support (FL-P) for different values of $N$ ($2^{10}$ and $2^{15}$) and $W$ (1, 10, 20). FL-P is configured with 8 partitions. The window size essentially defines the size of a transaction, i.e., how many callees have to be merged.

A first observation is that, for $W = 1$, FL-P has a higher single-threaded overhead than FL because of the extra indirection to select the partition has an high impact with short

Table 5.5: Contribution of helpers to the global commit throughput.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FL 3 thr.</th>
<th>FL 6 thr.</th>
<th>FL 8 thr.</th>
<th>FL 12 thr.</th>
<th>FL-P 3 thr.</th>
<th>FL-P 6 thr.</th>
<th>FL-P 8 thr.</th>
<th>FL-P 12 thr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N=1024$ $W=01$</td>
<td>41.8</td>
<td>54.7</td>
<td>50.3</td>
<td>49.4</td>
<td>13.7</td>
<td>27.5</td>
<td>33.4</td>
<td>42.7</td>
</tr>
<tr>
<td>$N=1024$ $W=10$</td>
<td>33.0</td>
<td>49.9</td>
<td>56.7</td>
<td>56.2</td>
<td>12.8</td>
<td>23.8</td>
<td>30.1</td>
<td>39.5</td>
</tr>
<tr>
<td>$N=32768$ $W=10$</td>
<td>32.9</td>
<td>49.9</td>
<td>53.5</td>
<td>55.0</td>
<td>12.0</td>
<td>24.0</td>
<td>30.7</td>
<td>40.2</td>
</tr>
<tr>
<td>$N=32768$ $W=20$</td>
<td>30.1</td>
<td>49.3</td>
<td>55.8</td>
<td>54.9</td>
<td>10.7</td>
<td>22.2</td>
<td>28.5</td>
<td>37.9</td>
</tr>
</tbody>
</table>
transactions. Except for that configuration, FL-P performs and scales systematically better than FL and other STMs thanks to having one master per partition. If more threads are active than partitions (12 threads vs. 8 partitions), FL-P is still able to scale because the threads execute transactions as helpers. This contribution of helper threads is shown in Table 5.5, which lists the percentage of commits achieved by FL and FL-P for 3, 6, 8, and 12 threads. Without partitioning, the contribution of helpers grows from 30% to 55%. When using partitions, it remains in the 12% to 42% range because more threads execute as master, but it continues to grow when using more threads than partitions.

Table 5.6: Abort rates for STMs and FASTLANE (12 threads).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Aborts (%)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WT</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ETL</td>
<td>1.5</td>
<td>3.0</td>
<td>0.1</td>
</tr>
<tr>
<td>NOREC</td>
<td>0.1</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>FL</td>
<td>0.6</td>
<td>0.9</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Note that using partitions with other STMs would not produce the same benefits as it would not suppress the overheads associated with transaction management (context saving, instrumentation). Table 5.6 shows the abort rates for all STMs and FASTLANE when 12 threads are active. TiNYSTM and NOREC have very low abort rates that would not diminish much if one instance per partition would be used; only the contention on their shared counter would be reduced. TML has a very high abort rate because almost all transactions perform updates and hence are practically serialized. Having one instance per partition would correspond to having a shared counter per partition, enabling parallelism as long as not two threads try to access the same partition. FL has an abort rate that increases with large transaction sizes because the fast master will invalidate slow helper transactions more often. Finally, FL-P has a moderate abort rate due to the fact that more threads are active than partitions. As compared to having a shared lock per partition, the speculative helpers can execute concurrently with masters and allow for more parallelism.

One can notice a drop in the throughput of FL-P after 6 threads for $W = 1$ and, to a lesser extend, for $W = 10$. As previously said, this is due to fact that when more than 6 threads are active, the second socket of our experimental machine is in use, which creates expensive cache coherence traffic over the interconnect. For longer transactions ($W = 20$), there is no noticeable degradation because the cache is less effectively used even on a single processor.

Interestingly, FL-P provides significant improvements over sequential with 2 threads already, and scales remarkably well, making the switch to STM less crucial even for 12 threads (except for $W = 1$ where ETL wins starting with 8 threads).

As a final remark, this benchmark is representative of applications with low latency requirements, e.g., for stream processing. Having one master per partition allows us to guarantee fast processing with almost predictable execution times, comparable to non-instrumented execution.

5.5 SUMMARY

In this chapter, we have addressed one of the main drawbacks of STM: its limited performance at low thread counts, as compared to the execution of the sequential original application without the overheads of TM.

We have proposed a novel synchronization strategy, FASTLANE, designed to perform best at low thread counts, where classical STM implementations are slower than sequential exe-
cution, which is typically between 2 and 4 threads. FASTLANE relies on a single pessimistic master thread with light instrumentation that never aborts, and one or more speculative helper threads that perform additional work as they try to commit their transactions without hampering the progress of the master. We have presented several variants that differ mainly by the optimizations they introduce in the commit function of the helper threads.

We have introduced a new benchmark that computes communities of interest, with a transactional workload that can be partitioned such that multiple masters can execute in distinct partitions. Results from experimental evaluation show that our new algorithms performs well in their target operation range of low thread counts, and provide a real performance boost if combined with partitioning.

Currently, the decision of when to change between the different code paths (sequential, FASTLANE, and STM) is based on the number of physical cores available on the target machine when starting the application. In the future, we would like to investigate dynamic schemes by periodically switching modes for short durations to gather samples of the commit rate and taking decision on than basis. This simple sampling-based approach could obviously be combined with more sophisticated strategies, e.g., using modeling-based techniques, depending on the workload’s nature.
6 THE TURBO DIARIES: APPLICATION-CONTROLLED FREQUENCY SCALING EXPLAINED*

*The contents of this chapter first appeared at USENIX ATC ’14 [Wam+14].
Most multicore architectures nowadays support dynamic voltage and frequency scaling (DVFS) to adapt their speed to the system’s load and save energy. Some recent architectures additionally allow cores to operate at boosted speeds exceeding the nominal base frequency but within their thermal design power.

In this chapter, we propose a general-purpose library that allows selective control of DVFS from user space to accelerate multi-threaded applications and expose the potential of heterogeneous frequencies. We analyze the performance and energy trade-offs using different DVFS configuration strategies on several benchmarks and real-world workloads. With the focus on performance, we compare the latency of traditional strategies that halt or busy-wait on contended locks and show the power implications of boosting of the lock owner. We propose new strategies that assign heterogeneous and possibly boosted frequencies while all cores remain fully operational. This allows us to leverage performance gains at the application-level while all threads continuously execute at different speeds. We also derive a model to help developers decide on the optimal DVFS configuration strategy, e.g., for lock implementations. Our in-depth analysis and experimental evaluation of current hardware provides insightful guidelines for the design of future hardware power management and its operating system interface.

6.1 INTRODUCTION

While early generations of multicore processors were essentially homogeneous with all cores operating at the same clock speed, new generations provide finer control over the frequency and voltage of the individual cores. A major motivation for this new functionality is to maximize processor performance without exceeding the thermal design power (TDP), as well as reducing energy consumption by decelerating idle cores [Bur+00; PLS01].

Two main CPU manufacturers, Intel and AMD, have proposed competing yet largely similar technologies for dynamic voltage and frequency scaling (DVFS) that can exceed the processor’s nominal operation frequency, respectively named Turbo Boost [Rot+12] and Turbo CORE [BFS12]. When the majority of cores are powered down or run at a low frequency, the remaining cores can boost within the limits of the TDP. In the context of multi-threaded applications, a typical use case is the optimization of sequential bottlenecks: waiting threads halt the underlying core and allow the owner thread to speed up execution of the critical section.

Boosting is typically controlled by hardware and is completely transparent to the operating system (OS) and applications. Yet, it is sometimes desirable to be able to finely control these features from an application as needed. Examples include: accelerating the execution of key sections of code on the critical path of multi-threaded applications [DSM13]; boosting time-critical operations or high-priority threads; or reducing the energy consumption of applications executing low-priority threads. Furthermore, workloads specifically designed to run on processors with heterogeneous cores (e.g., few fast and many slow cores) may take additional advantage of application-level frequency scaling. We argue that, in all these cases, fine-grained tuning of core speeds requires application knowledge and hence cannot be efficiently performed by hardware only.

Both Intel and AMD hardware implementations are constrained in several ways, e.g., some combination of frequencies are disallowed, cores must be scaled up/down in groups, or the CPU hardware might not comply with the scaling request in some circumstances. Despite the differences of both technologies, our comparative analysis derives a common abstraction for the processor performance states (Section 6.3). Based on the observed properties, we present the design and implementation of TURBO, a general-purpose library for application-level DVFS control that can programatically configure the speed of the cores of CPUs with AMD’s Turbo CORE and Intel’s Turbo Boost technologies, while abstracting the low-level differences and complexities (Section 6.4).
The cost of frequency and voltage transitions is subject to important variations depending on the method used for modifying processor states and the specific change requested. The publicly available documentation is sparse, and we believe to be the first to publish an in-depth investigation on the latency, performance, and limitations of these DVFS technologies (Section 6.5). Unlike previous research, our goal is not energy conservation or thermal boosting [Rag+13a], which is usually applied to mobile devices and interactive applications with long idle periods, but long running applications often found on servers. We target efficiency by focusing on the best performance, i.e., shorter run times or higher throughput using the available TDP. In this context, hardware is tuned in combination with the OS to use frequency scaling for boosting sequential bottlenecks on the critical path of multi-threaded applications. We use the TURBO library to measure the performance and power implications of both blocking and spinning locks (Section 6.5.2). Our evaluation shows that connecting knowledge of application behavior to programmatic control of DVFS confers great benefits on applications having heterogeneous load. We propose new configuration strategies that keep all cores operational and allow a manual boosting control (Section 6.5.3).

Based on the evaluation of manual configuration strategies and their latencies, we derive a simplified cost model (Section 6.5.4) to guide developers at which size of a critical region a frequency transition pays off. Four case studies investigate the performance gains exploited by application-level frequency control based on real-world benchmarks (Section 6.6).

6.2 RELATED WORK

The field of DVFS is dominated by work about improving energy efficiency [HM07; Kum+05; Now+02]. DVFS is proposed as a mid-term solution to the prediction that, in future processor generations, the scale of cores will be limited by power constraints [BC11; CSG11; Esm+11a]. In the longer term, chip designs are expected to combine few large cores for compute intensive tasks with many small cores for parallel code on a single heterogeneous chip. Not all cores can be active simultaneously due to thermal constraints [Kum+03; Ven+10]. A similar effect is achieved by introducing heterogeneous voltages and frequencies to cores of the same ISA [Em+03]. Energy efficiency is achieved by reducing the frequency and it was observed that the overall performance is only reduced slightly because it is dominated by memory [Lau+11] or network latencies.

Semeraro et al. [Sem+02] propose multiple clock domains with individual DVFS. Inter-domain synchronization is implemented using existing queues to minimize latency, and frequency can be reduced for events that are not on the application’s critical path. The energy savings can be extended by profile-based reconfiguration [Cai+08; Mag+03]. Another interesting approach to save power is to combine DVFS with inter-core prefetching of data into caches [KST12]. This can improve performance and energy efficiency, even on serial code, when more cores are active at a lower frequency. Choi et al. [CSP04] introduce a technique to decompose programs into CPU-bound (on-chip) and memory-bound (off-chip) operations. The decomposition allows fine tuning of the energy-performance trade-off, with the frequency being scaled based on the ratio of the on-chip to off-chip latencies. The energy savings come with little performance degradation on several workloads running on a single core. Hsu et al. [HF05] propose an algorithm to save energy by reducing the frequency with HPC workloads. Authors also present and discuss transition latencies. A recent study [Lar+12] on the Cray XT architecture, which is based on AMD CPUs, demonstrates that significant power savings can be achieved with little impact on runtime performance when limiting both processor frequency and network bandwidth. The P-states are changed before the application runs. It is recommended that future platforms provide DVFS of the different system components to exploit the trade-offs between energy and performance. Our work goes in the same direction, by investigating the technical means to finely control the states of individual cores.
While energy efficiency has been widely studied, few researchers have addressed DVFS to speed up workloads [HM08]. Park et al. [Par+13] present a detailed DVFS transition overhead model based on a simulator of real CPUs. For a large class of multi-threaded applications, an optimal scheduling of threads to cores can significantly improve performance [Rag+13b]. Isci et al. [Isc+06] propose using a lightweight global power manager for CPUs that adapts DVFS to the workload characteristics. Suleman et al. [Sul+09] optimize the design of asymmetric multicores for critical sections. A study of Turbo Boost has shown that achievable speedups can be improved by pairing CPU intensive workloads to the same core [Cha+09]. This allows masking delays caused by memory accesses. Results show a correlation between the boosting speedup and the LLC miss rate (high for memory-intensive applications). DVFS on recent AMD processors with a memory-bound workload limits energy efficiency because of an increase of static power in lower frequencies/voltages [LH10]. Ren et al. [Ren+13] investigate workloads that can take advantage of heterogeneous processors (fast and slow) and show that throughput can be increased by up to 50% as compared with using homogeneous cores. Such workloads represent interesting use cases for DVFS.

Our TURBO library complements much of the related work discussed in this section, in that it can be used to implement the different designs and algorithms proposed in these papers.

## 6.3 Hardware Support for Boosting

With both AMD’s Turbo CORE and Intel’s Turbo Boost, performance levels and power consumption of the processor are controlled through two types of operational states: P-states implement DVFS and set different frequency/voltage pairs for operation, trading off higher voltage (and thus higher power draw) with higher performance through increased operation frequency. P-states can be controlled through special machine-specific registers (MSRs) that are accessed through the `rdmsr/wrmsr` instructions. The OS can request a P-state change by modifying the respective MSR. P-state changes are also not instantaneous: the current needs to be adapted and frequencies are ramped, both taking observable time.

C-states are used to save energy when a core is idle. C0 is the normal operational state. All other C-states halt the execution of instructions and trade different levels of entry/wakeup latency for lower power draw. The OS can invoke C-states through various means such as the `hlt` and `monitor/mwait` instructions. We argue in this chapter that there are benefits in keeping selected cores operational, albeit at a lower frequency, and that manipulating P-states can be more efficient in terms of latency than manipulating C-states.

We base our work on AMD’s FX-8120 [AMD12] and Intel’s i7-4770 [Int14] CPUs, whose characteristics are listed in Table 6.1.

### Table 6.1: Specification of the AMD and Intel processors.

<table>
<thead>
<tr>
<th>AMD FX-8120</th>
<th>Intel i7-4770</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td><strong>Intel Core 4th generation</strong></td>
</tr>
<tr>
<td><strong>Codename</strong></td>
<td>&quot;Haswell&quot;</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td>4 cores with hyper-threading</td>
</tr>
<tr>
<td><strong>L2 cache</strong></td>
<td>4x256KB per core</td>
</tr>
<tr>
<td><strong>L3 cache</strong></td>
<td>1x8MB per package</td>
</tr>
<tr>
<td><strong>TDP</strong></td>
<td>124.95W (NB 14.23W)</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>3.4GHz (0.8–3.9GHz)</td>
</tr>
<tr>
<td><strong>Stepping</strong></td>
<td>ACPI P-states, 100MHz</td>
</tr>
<tr>
<td><strong>Voltage</strong></td>
<td>0.707–1.86V (5–75W)</td>
</tr>
<tr>
<td></td>
<td>1.875–1.412V (3.41–27.68W)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.3.1 AMD’S TURBO CORE

The architecture of the AMD FX-8120 processor is illustrated in Figure 6.1. The cores of a package are organized by pairs in modules that share parts of the logic between the two cores.

Our processor supports seven P-states summarized in Table 6.2. We introduce a Turbo naming convention to abstract from the manufacturer specifics. AMD uses P-state numbering based on the ACPI standard with P0 being the highest performance state. The two topmost are boosted P-states (#P\text{boosted} = 2) that are by default controlled by the hardware. The remaining five P-states can be set by the OS through the MSRs\(^1\).

The boosting of the frequency beyond the nominal P-state (P\text{base}) is enabled by the hardware’s Turbo CORE technology if operating conditions permit. The processor determines the current power consumption and will enable the first level of boosting (P1\text{HW}) if the total power draw remains within the TDP limit and the OS requests the fastest software P-state. A multi-threaded application can boost one module to P1\text{HW} while others are in P\text{base} if it does not use all features of the package to provide the required power headroom, e.g., no FPUs are active. The fastest boosting level (P\text{turbo}) is entered automatically if some cores have furthermore reduced their power consumption by entering a deep C-state. Note that Turbo CORE is deterministic, governed only by power draw and not temperature, such that the maximum frequency is workload dependent. During a P-state transition, the processor remains active and capable of executing instructions, and the completion of a P-state transition is indicated in an MSR available to the OS.

The Turbo CORE features can be enabled or disabled altogether, i.e., no core will run above P\text{base}. Additionally, selected AMD processors allow developers to control the number of hardware-reserved P-states by changing #P\text{boosted} through a configuration MSR. To achieve manual control over all P-states, including boosting, one can set #P\text{boosted} = 0. The core safety mechanisms are still in effect: the hardware only enters a boosted P-state if the TDP limit

\(^1\)The numbering in software differs from the actual hardware P-states: P\text{HW} = P\text{SW} + #P\text{boosted}. With a default of #P\text{boosted} = 2: P_{\text{base}} = P_{0\text{SW}} = P_{2\text{HW}} and P_{\text{turbo}} = P_{0\text{HW}}. P_{0\text{SW}} is the fastest requestable software P-state.
has not been reached. In contrast to the processor’s automatic policy, the manual control of all P-states can enable \( P_{\text{turbo}} \) with all other cores in C0 but running at \( P_{\text{slow}} \).

Due to the pairwise organization of cores in modules, the effect of a P- and C-state change depends on the state of the sibling core. While neighboring cores can request P-states independently, the fastest selected P-state of the two cores will apply to the entire module. Since the \texttt{wrmsr} instruction can only access MSRs of the current core, it can gain full control over the frequency scaling if the other core is running at \( P_{\text{slow}} \). A module only halts if both cores are not in C0.

The processor allows to read the current power draw (\( P \)) that it calculates based on the load. Out of the total TDP, 14.24W are reserved for the northbridge (NB) (including L3 cache) and logic external to the cores. Each of the four modules is a voltage (\( V \)) and frequency (\( f \)) domain defined by the P-state. The package requests \( V \) defined by the fastest active P-state of any module from the voltage regulator module (VRM). Table 6.2 lists \( P \) with (1) all cores in the same P-state executing \texttt{nop} instructions, (2) execution of integer operations with \texttt{ALU}, (3) three modules in \( P_{\text{slow}} \) except one in the given P-state, and (4) all modules halted using \texttt{mwait} except one active core. The consumed active \( P \) depends on \( V \), \( f \) and the capacitance (\( C \)) that varies dynamically with the workload (\( P = V^2 \cdot f \cdot C_{\text{dyn}} \)). Therefore, for the \texttt{nop} load all cores can boost to \( P_{1 \text{HW}} \), while for integer loads all cores can run only at \( P_{\text{base}} \). Boosting under load can be achieved when other modules are either in \( P_{\text{slow}} \) or halted. \texttt{mwait} provides the power headroom to automatically boost to \( P_{\text{turbo}} \). The manual boosting control allows to run one module in \( P_{\text{turbo}} \) if the others run at \( P_{\text{slow}} \).

### 6.3.2 INTEL'S TURBO BOOST

Intel’s DVFS implementation is largely similar to AMD’s but more hardware-centric and mainly differs in the level of manual control. All cores are in the same frequency and voltage domain but can have an individual C-state. The P-states are based on increasing multipliers for the stepping of 100MHz, non-predefined ACPI P-states in the opposite order. Our processor supports frequencies from 0.8GHz to 3.9GHz corresponding to 32 P-states that are summarized in Table 6.3. In \texttt{TURBO} terms, \( P_{\text{base}} \) corresponds to \( P_{34\text{HW}} \), leaving 5 boosted P-states. All active cores in C0 symmetrically run at the highest requested frequency, even if some cores requested slower P-states. The consumed power was measured in a fashion analogous to that in Section 6.3.1, with hyper-threading enabled and all cores always in the same P-State.

The processor enables Turbo Boost if not all cores are in C0. The level of boosting depends on the number of active cores, estimated power consumption, and additionally the temperature of the package. This “thermal boosting” allows the processor to temporarily exceed the TDP using the thermal capacitance of the package. In contrast to AMD, the maximum achievable frequency also depends on the recent execution history, which relates to the current package temperature and makes it somewhat stateful. While boosting can be enabled or disabled altogether, the boosted P-states are always controlled automatically by the processor.
Intel’s design choice has the goal to speed up critical periods of computation, e.g., boosting sequential bottlenecks by putting waiting cores to sleep using C-states or providing temporarily high performance for interactive applications found on mobile devices or desktops. Our focus is on multi-threaded applications mostly found on servers that run for long periods without much idle time. Thermal boosting is not applicable to such workloads because on average one cannot exceed the TDP. Instead, our goal is to improve the performance within the TDP limits.

### 6.4 TURBO LIBRARY

The TURBO library, written in C++ for the Linux OS, provides components to configure, control, and profile processors from within applications. Our design goals are twofold: we want to provide a set of abstractions to (1) make it convenient to improve highly optimized software based on DVFS; and (2) set up a testbed for algorithms that explore challenges of future heterogeneous cores [BC11], such as schedulers. The components of the TURBO library are organized in layers with different levels of abstraction as shown in Figure 6.2. All components can be used individually to support existing applications that use multiple threads or processes. The layered architecture allows an easy extension to future hardware and OS revisions.

#### 6.4.1 PROCESSOR AND LINUX KERNEL SETUP

The default configurations of the processors and Linux kernel manage DVFS transparently for applications: All boosted P-states are controlled by the processor and the Linux governor will adapt the non-boosted P-states based on the current processor utilization (“ondemand”) or based on static settings that are enforced periodically (“performance”, “userspace”).

We must disable the influence of the governors and the processor’ power saving features in order to gain explicit control of the P-states and boosting in user space using our library. Note that the “userspace” governor provides an alternative but inefficient P-state interface [Hil+13].
Therefore, we disable the CPU frequency driver (cpufreq) and turn off AMD’s Cool’n’Quiet speed throttling technology in the BIOS. To control all available P-states in user space, we can either disable automatic boosting altogether, which is the only solution for Intel, or for AMD set \#P\text{boosted} = 0 to enable manual boosting control (for details see Section 6.3). Changing the number of boosted P-states also changes the frequency of the time stamp counter (tsc) for AMD processors so we therefore disable tsc as a clock source for the Linux kernel and instead use the high precision event timer (hpet). Note that these tweaks can easily be applied to production systems because we only change BIOS settings and kernel parameters.

The processor additionally applies automatic frequency scaling for the integrated NB that can have a negative impact on memory access times for boosted processor cores. Therefore, NB P-states are disabled and it always runs at the highest possible frequency.

Linux uses the monitor and mwait instructions to idle cores and change their C-state. When another core writes to the address range specified by monitor, then the core waiting on mwait wakes up. The monitor-mwait facility provides a “polite” busy-waiting mechanism that minimizes the resources consumed by the waiting thread. For experiments on AMD, we enable these processor instructions for user space and disable the use of mwait in the kernel to avoid lockouts. Similarly, we must also disable the use of the hlt instruction by the kernel, because otherwise we cannot guarantee that at least one core stays in C0. We restrict the C-state for the Linux kernel to C0 and use the polling idle mode. These changes are required in our prototype only for the evaluation of C-state transitions and are not necessary in a production system.

The presented setup highlights the importance of the configuration of both hardware and OS for sound benchmarking. Multi-threaded algorithms should be evaluated by enforcing P\text{base} and C0 on all cores to prevent inaccuracies due to frequency scaling and transition latencies. All other sources of unpredictability should be stopped, e.g., all periodic cron jobs.

### 6.4.2 PERFORMANCE CONFIGURATION INTERFACE

The library must be aware of all threads even if they are managed explicitly by the application. Therefore, the thread registry is used first to create or register all threads. Next, the threads are typically assigned to distinct cores based on the processor’s topology, which is discovered during initialization. If thread migration to another core is required at runtime, it must be performed using our library to allow an update of the core specific configuration, e.g., the P-state.

The easiest way to benefit from DVFS is to replace the application’s locks with thread control wrappers that are decorated with implicit P-state transitions, e.g., boosting the lock owner at P\text{turbo}, waiting at P\text{slow}, and executing parallel code at P\text{base}.

If the wrappers are not sufficient, the application can request an explicit performance configuration that is still independent of the underlying hardware. Threads can request the executing core to run at P\text{turbo}, P\text{base}, or P\text{slow}, and can alternatively specify the P-state in percent based on the maximum frequency. The actual P-state is derived from the selected setup, e.g., if boosting is enabled and controlled manually. The current P-state configuration is cached in the library in order to save the overheads from accessing the MSRs in kernel space. If a P-state is requested that is already set or cannot be supported by the processor’s policy or TDP limits, then the operation has no effect.\(^2\) Threads can also request to temporarily migrate to a dedicated processor core that runs at the highest possible frequency and stays fully operational in C0.

\(^2\)In practice, we write our request in MSR\(\_\text{P\_cmd}\) and can read from MSR\(\_\text{P\_val}\) what the CPU actually decided. We can either (a) wait until both MSRs match, i.e., another core makes room in the TDP, (b) return the CPU’s decision, or (c) just write and provide best-effort guarantees (default). Deterministic hardware without thermal boosting does not overwrite MSR\(\_\text{P\_cmd}\).
The lowest layer presents **hardware abstractions** for the machine specific interfaces and DVFS implementations, as well as the Linux OS. The Linux kernel provides a device driver that lets applications access MSRs as files under root privilege using `pread` and `pwrite`. We implemented a lightweight TURBO kernel driver for a more streamlined access to the processor’s MSRs using `ioctl` calls. The driver essentially provides a wrapper for the `wrmsr/rdmsr` instructions to be executed on the current core. Additionally, it allows kernel space latency measurements, e.g., for P-state transition time, with more accuracy than from user space. We derive the **topology** from the Linux ACPI driver and use `sysfs` for AMD’s package configuration using PCI functions.

### 6.4.3 PERFORMANCE AND POWER PROFILING

The TURBO library provides means to profile highly optimized applications and algorithms for heterogeneous cores. The profiling can be used to first identify sections that can benefit from frequency scaling and later to evaluate the performance and power implications of different configurations.

Again, the simplest ways to obtain statistics is to use **thread control wrappers**, which exist to replace locks, barriers, and condition variables. The wrappers can be decorated with profiling capabilities of the **performance monitor**, which uses the `aperf/mperf` and `tsc` counters of the processor [AMD12; Int14] and the `perf_event` facilities of the Linux kernel to access the processor’s **performance monitoring unit (PMU)**.

The performance monitor operates in intervals, e.g., defined by a lock wrapper, for which it captures the cycles, frequency, and C-state transitions. Additional counters such as the number of cache misses or stalled cycles can be activated, e.g., to analyze the properties of a critical section. The PMU also provides counters to read the **running average power limit (RAPL)** on Intel and the processor power in TDP on AMD.

### 6.5 PROCESSOR EVALUATION

On top of the TURBO library presented in Section 6.4, we implemented a set of benchmark applications that configure and profile the underlying processor. In this section, we present (1) the static transition latencies introduced by the OS and hardware, (2) the overheads of blocking upon contended locks and when it pays off regarding speed and energy compared to spinlocks, and (3) new static and dynamic P-state transition strategies that optimize spinlocks and allow applications to expose heterogeneous frequencies.

#### 6.5.1 HARDWARE TRANSITION LATENCY

The latency for DVFS results from a combination of OS overhead to initiate a transition and hardware latency to adjust the processor’s state. Therefore, we present in Tables 6.4 (AMD) and 6.5 (Intel) the overhead for system calls, P-state requests and the actual transition latencies in isolation. Throughout our evaluation, we use a Linux kernel 3.11 that is configured according to Section 6.4.1. We use only the x86 cores (ALU) and no FPU or MMX/SSE/AVX to preserve the required headroom for manual boosting.

System calls for device-specific input/output operations (`ioctl`) have a low overhead and are easily extensible using the request code parameter. The interface of the TURBO driver (`trb`) is based on `ioctl`, while the Linux MSR driver (`msr`) uses a file-based interface that can be accessed most efficiently using `pread/pwrite`. The difference in speed between `msr` and `trb` (both use `rdmsr/wrmsr` to access the MSRs) results mostly from additional security checks and indirections that we streamlined for the TURBO driver. The cost in time for system calls depends on the P-state, i.e., reading the current P-state scales with the selected frequency, here $P_{base}$. 

---

104
6.5 Processor Evaluation

<table>
<thead>
<tr>
<th>Operation</th>
<th>P-State Transition</th>
<th>Mean</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Cycles</td>
<td>ns</td>
</tr>
<tr>
<td>syscall(futex_wait_private)</td>
<td>—</td>
<td>1321</td>
<td>330</td>
</tr>
<tr>
<td>ioctl(trb)</td>
<td>—</td>
<td>920</td>
<td>230</td>
</tr>
</tbody>
</table>

P-state MSR read/write cost using msr or TURBO driver

<table>
<thead>
<tr>
<th>Operation</th>
<th>P-State Transition</th>
<th>Mean</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Cycles</td>
<td>ns</td>
</tr>
<tr>
<td>pread(msr, pstate)</td>
<td>—</td>
<td>3044</td>
<td>761</td>
</tr>
<tr>
<td>ioctl(trb, pstate)</td>
<td>—</td>
<td>2299</td>
<td>574</td>
</tr>
<tr>
<td>pwrite(msr, pstate, P&lt;sub&gt;base&lt;/sub&gt;)</td>
<td>P&lt;sub&gt;base&lt;/sub&gt;→P&lt;sub&gt;base&lt;/sub&gt;</td>
<td>2067</td>
<td>741</td>
</tr>
<tr>
<td>ioctl(trb, pstate, P&lt;sub&gt;base&lt;/sub&gt;)</td>
<td>P&lt;sub&gt;base&lt;/sub&gt;→P&lt;sub&gt;base&lt;/sub&gt;</td>
<td>1875</td>
<td>468</td>
</tr>
</tbody>
</table>

Hardware latencies for P-state set (wrmsr) and transition (wait) (kernel space)

<table>
<thead>
<tr>
<th>Operation</th>
<th>P-State Transition</th>
<th>Mean</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Cycles</td>
<td>ns</td>
</tr>
<tr>
<td>wrmsr(pstate, P&lt;sub&gt;slow&lt;/sub&gt;)</td>
<td>P&lt;sub&gt;base&lt;/sub&gt;→P&lt;sub&gt;slow&lt;/sub&gt;</td>
<td>28087</td>
<td>7021</td>
</tr>
<tr>
<td>wrmsr(pstate, P&lt;sub&gt;slow&lt;/sub&gt;) &amp; wait</td>
<td>P&lt;sub&gt;base&lt;/sub&gt;→P&lt;sub&gt;slow&lt;/sub&gt;</td>
<td>29783</td>
<td>7445</td>
</tr>
<tr>
<td>wrmsr(pstate, P&lt;sub&gt;turbo&lt;/sub&gt;)</td>
<td>P&lt;sub&gt;slow&lt;/sub&gt;→P&lt;sub&gt;turbo&lt;/sub&gt;</td>
<td>1884</td>
<td>471</td>
</tr>
<tr>
<td>wrmsr(pstate, P&lt;sub&gt;turbo&lt;/sub&gt;) &amp; wait</td>
<td>P&lt;sub&gt;slow&lt;/sub&gt;→P&lt;sub&gt;turbo&lt;/sub&gt;</td>
<td>226988</td>
<td>56747</td>
</tr>
<tr>
<td>wrmsr(pstate, P&lt;sub&gt;base&lt;/sub&gt;) &amp; wait</td>
<td>P&lt;sub&gt;slow&lt;/sub&gt;→P&lt;sub&gt;base&lt;/sub&gt;</td>
<td>183359</td>
<td>45839</td>
</tr>
<tr>
<td>wrmsr(pstate, P&lt;sub&gt;turbo&lt;/sub&gt;) &amp; wait</td>
<td>P&lt;sub&gt;base&lt;/sub&gt;→P&lt;sub&gt;turbo&lt;/sub&gt;</td>
<td>94659</td>
<td>23664</td>
</tr>
</tbody>
</table>

Hardware latencies for C-state transitions (in user space)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mean</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>ns</td>
</tr>
<tr>
<td>monitor &amp; mwait</td>
<td>1818</td>
<td>454</td>
</tr>
</tbody>
</table>

Software and hardware latency for thread migration

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mean</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_setaffinity</td>
<td>26728</td>
<td>6682</td>
</tr>
</tbody>
</table>

Table 6.4: Latency cost (AMD FX-8120, 100,000 runs).

Observation 1. P-state control should be made available through platform-independent application program interfaces (APIs) or unprivileged instructions. The latter would additionally eliminate the latency for switching into kernel space to access platform-specific MSRs but require that the OS’s DVFS is disabled.

We measured the cost of the wrmsr instruction that initiates a P-State transition of the current core, as well as the latency until the transition is finished, by busy waiting until the frequency identifier of the P-state is set in the status MSR. Both measurements are performed in the TURBO driver, removing the inaccuracy due to system call overheads.

For AMD, requesting a P-state faster than the current one (e.g., P<sub>slow</sub>→P<sub>base</sub>) has low overhead in itself, but the entire transition has a high latency due to the time the VRM takes to reach the target voltage. The request to switch to a slower P-state (e.g., P<sub>base</sub>→P<sub>slow</sub>) has almost the same latency as the entire transition, i.e., the core is blocked during most of the transition. We suspect that this blocking may be caused by a slow handshake to coordinate with the other module’s core to see if an actual P-state change will occur. Overall, the transition has a lower latency because the frequency can already be reduced before the voltage regulator is finished. If only switching to a slow P-state for a short period, the transition to a faster P-state will be faster if the voltage was not dropped completely.

On the Intel CPU, total latency results are very similar: A P-state transition also takes tens of microseconds but depends on the distance between the current and requested P-state. A significant difference to AMD, however, lies in the faster execution of the wrmsr request of a P-state transition going slower (e.g., P<sub>base</sub>→P<sub>slow</sub>) because Intel does not need to perform
additional coordination.

**Observation 2.** The frequency transitions should be asynchronous, triggered by a request and not blocking, i.e., keeping the core operational. The API should include the ability to read or query P-state transition costs for building a cost model that allows DVFS-aware code to adapt at runtime.

We additionally show costs related to the OS. In the *mwait* experiment, one core continuously updates a memory location while the other core specifies the location using *monitor* and calls *mwait*. The core will immediately return to execution because it sees the memory location changed, so the numbers represent the minimal cost of executing both instructions. Although AMD allows the use of *mwait* from user space, the feature is typically used by the OS’s *futex* system call when the kernel decides to idle. The *pthread_setaffinity* function migrates a thread to a core with a different L2 Cache that is already in C0 state and returns when the migration is finished. Thread migration typically results in many cache misses but the benchmark keeps only minimal data in the cache.

**Observation 3.** The OS should keep the current frequency in the thread context to better support context switches and thread migrations. Ideally, the OS would expose a new set of advisory platform-independent APIs to allow threads to set their desired DVFS-related performance targets. Furthermore, the OS kernel (and potentially a virtual machine hypervisor) would moderate potentially conflicting DVFS resource requests from independent and mutually unaware applications.
6.5.2 BLOCKING VS. SPINNING LOCKS

We evaluate the boosting capabilities using a thread on each core that spends all its time in critical sections (CS). The CS is protected by a single global lock implemented as an MCS queue lock [MS91] in the TURBO library. The lock is decorated such that upon contention, the waiting thread either spins or blocks using mwait (AMD only) or futex. The sequence is illustrated in Figure 6.4a and 6.4b, respectively. In all cases, the thread-local MCS node is used for the notification of a successful lock acquisition. Inside the CS, a thread-local counter is incremented for a configurable number of iterations (∼10 cycles each). While the global lock prevents any parallelism, the goal of the concurrent execution is to find the CS length that amortizes the DVFS cost.

We want to discuss when blocking is preferable over spinning, both in terms of performance and energy, using the default configuration of hardware and OS: The P-states are managed automatically by the processor and the setup from Section 6.4.1 is not applied. We run the application for 100 seconds and count the number of executed CS, which gives us the cycles per CS including all overheads. Separately, we measure the cycles per CS without synchronization at Pbase, i.e., the cycles doing real work. The effective frequency inside a CS is:

\[ f_{CS} = f_{base} \cdot \frac{\text{cycles}_{nosync}}{\text{cycles}_{mcs}} \]

The energy results are based on the processor’s TDP/RAPL values, from which we take samples during another execution. We compute the energy it takes to execute 1 hour of work at Pbase inside CS:

\[ E = E_{sample} \cdot \frac{\text{cycles}_{mcs}}{\text{cycles}_{nosync}} \]

The results are shown in Figure 6.3. The spin strategy runs all cores at Pbase and is only affected by synchronization overhead, with decreasing impact for larger sizes of CS. The mwait and futex strategies are additionally effected by C-state transitions that halt the core while blocking, which allows to boost the active core. The C-state reached by mwait is not deep enough to enable P_turbo, probably because it is requested from user space. Still, CS are executed at P1_{HW} and the low overhead lets mwait outperform spin already at a CS size of ∼4k cycles. Using futex has the highest overhead because it is a system call. The C-state reached depends on twait (see Figure 6.4b), which explains the performance drop: Deep C-states introduce a high latency (see Table 6.5) but are required to enable P_turbo. We verified this behavior using aperf/mpcrf, which showed that the frequency in C0 is at P_turbo only after the drop. The futex outperforms spin and mwait at ∼1.5M cycles for AMD and ∼4M cycles for Intel.

Figure 6.3: Characteristics of blocking and spinning.
Intel, which also boosts spin 2 steps. Note that an optimal synchronization strategy for other workloads also depends on the conflict probability and $t_{wait}$, but our focus is on comparing boosting initiated by the processor and on application-level.

The sampled power values do not vary for different sizes of CS (see Tables 6.2 and 6.3 for ALU and $m_{wait}$), except for $futex$, which varies between 55-124W for AMD depending on the reached C-state. The reduction in energy consumption due to deeper C-states must first amortize the introduced overhead before it is more efficient than spinning. With only a single core active at a time, $futex$ is the most energy efficient strategy for AMD after a CS size of $\sim$1M cycles, which results for 8 threads in $t_{wait} = \sim$7M cycles because the MCS queue lock is fair. Intel is already more energy efficient after $\sim$10k cycles, indicating that it trades power savings against higher latencies. Boosting provides performance gains for sequential bottlenecks and halting amortizes the active cores’ higher energy consumption [Miy+02]. The default automatic boosting is not energy efficient for scalable workloads because all energy is consumed only by a single core without performance benefit [Esm+11b].

### 6.5.3 APPLICATION-LEVEL P-STATE TRANSITION STRATEGIES

Our goal is to enable application-level DVFS while keeping all cores active. Therefore, we enable manual P-state control with the setup described in Section 6.4.1 and restrict the following discussion to just AMD. For the evaluation, we use the same application as in the previous Section 6.5.2 but with a different set of decorations for the lock: The strategy one executes iterations only on a single core that sets the P-state statically during initialization to either $P_{slow}$, $P_{base}$ or $P_{turbo}$. All other threads run idle on cores at $P_{slow}$ in C0. This provides the baseline for different P-state configurations without P-state transition overheads but includes the synchronization. The dynamic strategies ownr and wait are illustrated in Figure 6.4c. For ownr, all threads are initially set to $P_{slow}$ and the lock owner dynamically switches to $P_{turbo}$ during the CS. For wait, all threads initially request $P_{turbo}$ and dynamically switch to $P_{slow}$ while waiting. The processor prevents an oversubscription and allows $P_{turbo}$ only if 3 modules are in $P_{slow}$. The remaining strategies use only a subset of the cores for executing CS: dlgt uses only 1 thread per module and delegates the P-state transition request to the thread executing on the neighboring core. The strategy is otherwise the same as ownr. mgrt uses only 6 cores on 3 modules running at $P_{slow}$. The remaining module runs at $P_{turbo}$ and the current lock owner migrates to a core of the boosted module during the CS.

The results are presented in Figure 6.5. The dynamic strategies ownr and wait introduce overhead in addition to the synchronization costs because two P-state transitions must be requested for each CS. This overhead is amortized when the resulting effective frequency of the CS is above one with $P_{base}$, starting at CS sizes of $\sim$600k cycles. Both strategies behave similarly because the application does not execute parallel code between CS. Otherwise, the idea is that wait hides the slow blocking transition to $P_{slow}$ (see Section 6.5.1) within $t_{wait}$, whereas ownr must perform this transition after releasing the lock. To that extent, dlgt shifts the P-state transition cost entirely to the other core of the module and can outperform...
6.5 Processor Evaluation

Observation 4. The P-state transition should be as fast as possible so that short boosted sections can already amortize the transition cost. It exists hardware that can switch to arbitrary frequencies within one clock cycle [Hop+13].

As long as one module runs at P\textsubscript{turbo}, which is the case here, the processor consumes the maximal TDP of 125W. The consumed energy solely depends on the overheads of each strategy because of the serialized execution. Note that the energy for executing one with a static P-state is almost identical for P\textsubscript{slow}, P\textsubscript{base} and P\textsubscript{turbo}, indicating that the energy consumption is proportional to the P-state. In fact, we get for a single module in P\textsubscript{turbo} 29% more speed using 25% more power compared to P\textsubscript{base} (see Table 6.2). Compared to \texttt{mwait} and \texttt{futex}, application-level DVFS allows less power savings because all cores stay in C0, but it can be applied to parallel workloads, which we investigate in Section 6.6.

Observation 5. Processors should support heterogeneous frequencies individually for each core to provide headroom for boosting while staying active. The design should not limit the frequency domain for a package (Intel) or module (AMD). An integrated VRM supports fine-grained voltage domains to allow higher power savings at low speeds. Additionally, for some workloads it would be beneficial to efficiently set remote cores to P\textsubscript{slow} in order to have local boosting control.

6.5.4 PERFORMANCE COST MODEL

Based on our experimental results, we derive a simplified cost model for AMD’s boosting implementation to guide developers when boosting pays off regarding performance. We first present a model for boosting sequential bottlenecks that formalizes the results from Section 6.5.3. We then specialize it for boosting CS that are not a bottleneck as well as for workloads that contain periods with heterogeneous workload distributions.

We make the following simplifying assumptions: (1) the application runs at a constant rate of instructions per cycle (IPC), regardless of the processor frequency; (2) we do not consider
costs related to thread synchronization; (3) the frequency ramps linearly towards faster P-states (e.g., \( f_{P_{\text{slow}}} \rightarrow f_{P_{\text{turbo}}} \)); and (4) the frequency transition to a slower P-state takes as long as the P-state request. Assumption (4) is a direct result of our latency measurement, (1) and (2) allow an estimation without taking application specifics into account. We will revisit assumptions (1) and (2) when looking at actual applications that depend on memory performance and thus exhibit varying IPC with changing frequency (due to the changed ratio of memory bandwidth, latency and operation frequency).

For sequential bottlenecks, we follow the strategy described in Section 6.5.3 and illustrated in Figure 6.4c. Boosting will pay off if we outperform the CS that runs at \( f_{P_{\text{base}}} \):

\[
t_{CS_{P_{\text{turbo}}}} \leq t_{CS_{P_{\text{base}}}}
\]

The minimal \( t_{CS} \) must be greater than the combined P-state request latencies and the number of cycles that are executed during the P-State transition \( (t_{\text{ramp}}) \) to \( P_{\text{turbo}} \): The minimal \( t_{CS} \) must be greater than the combined P-state request latencies and the number of cycles that are executed during the P-State transition \( (t_{\text{ramp}}, \text{i.e., the difference between } w_{\text{msr}} \text{ and } w_{\text{wait}} \text{ in Table 6.4}) \) to \( P_{\text{turbo}} \):

\[
t_{CS} \geq t_{P_{\text{slow}} \rightarrow P_{\text{turbo}}} + t_{\text{ramp}} + t_{P_{\text{turbo}} \rightarrow P_{\text{base}}} + \frac{\text{cycles}_{CS} - \text{cycles}_{ramp}}{f_{P_{\text{turbo}}}}
\]

Based on the P-state transition behavior that we observed in Section 6.5.3, we can compute the minimal \( t_{CS} \) as follows:

\[
t_{CS} \geq \frac{f_{P_{\text{turbo}}}}{f_{P_{\text{turbo}}} - f_{P_{\text{base}}}} \cdot (t_{P_{\text{slow}} \rightarrow P_{\text{turbo}}} + t_{P_{\text{turbo}} \rightarrow P_{\text{base}}}) + \frac{1}{2} \cdot \frac{f_{P_{\text{turbo}}} - f_{P_{\text{slow}}}}{f_{P_{\text{turbo}}} - f_{P_{\text{base}}}} \cdot t_{\text{ramp}}
\]

The minimal wait time \( t_{\text{wait}} \) to acquire the lock should simply be larger than the time to drop to \( f_{P_{\text{slow}}} \): \( t_{\text{wait}} \geq t_{P_{\text{base}} \rightarrow P_{\text{slow}}} \). With the results from Section 6.5.1, on AMD this equals to a minimal \( t_{CS} \) of \( \sim 436,648 \) cycles (\( \sim 109 \mu s \)). Note that optimized strategies can reach the break even point already earlier (e.g., \( d_{\text{igf}} \) in Figure 6.5). Based on the above cost model for sequential bottlenecks, we can derive a cost model for boosting CS by one step (see Figure 6.4d):

\[
t_{CS} \geq \frac{f_{P_{\text{HW}}}^{1}}{f_{P_{\text{HW}}}^{1} - f_{P_{\text{base}}}} \cdot (t_{P_{\text{base}} \rightarrow P_{\text{HW}}}^{1} + t_{P_{\text{HW}}}^{1} \rightarrow P_{\text{base}}) + \frac{1}{2} \cdot t_{\text{ramp}}
\]

We never move below \( P_{\text{base}} \) and boosting pays off if \( t_{CS} \) is longer than \( \sim 336,072 \) cycles (\( \sim 84 \mu s \)).

Besides boosting sequential bottlenecks, another interesting target are periods of heterogeneous workload distributions. These workloads can run one thread temporarily at a higher priority than other active threads or have an asymmetric distribution of accesses to CS from threads. Typically, such critical regions are longer because they combine several CS, thus improving the chances of amortizing the transition cost. Based on the presented cost model, we compute the minimal duration of such periods instead of the CS size. We present examples in Section 6.6.

### 6.6 Boosting Applications

We evaluated the TURBO library using several real-world applications with user space DVFS on the AMD FX-8120. We chose these workloads to (1) validate the results from our synthetic benchmarks and the cost model to boost sequential bottlenecks; (2) highlight gains by using application knowledge to assign heterogeneous frequencies; (3) show the trade-offs when the IPC depends on the core frequency, e.g., due to memory accesses; and (4) outweigh the latency cost of switching P-states by delegating critical sections to boosted cores.
6.6 Boosting Applications

6.6.1 PYTHON GLOBAL INTERPRETER LOCK

The Python Global Interpreter Lock (GIL) is a well known sequential bottleneck based on a blocking lock. The GIL must always be owned when executing inside the interpreter. Its latest implementation holds the lock by default for a maximum of 5ms and then switches to another thread if requested. We are interested in applying some of the P-state configuration strategies presented in Section 6.5.3 to see if they provide practical benefits. For this evaluation, we use the ccbench application that is included in the Python distribution (version 3.4a).

The benchmark includes workloads that differ in the amount of time they spent holding the GIL: (1) the Pi calculation is implemented entirely in Python and spends all its time in the interpreter; (2) the computation of regular expressions (Regex) is implemented in C with a wrapper function that does not release the GIL; and (3) the bz2 compression and SHA1 hashing have wrappers for C functions that release the GIL, so most time is spent outside the interpreter. Table 6.6 summarizes the characteristics of the workloads.

We evaluate the following P-state configuration strategies in Figure 6.6. Base runs at \( P_{\text{base}} \) and, hence, does not incur P-state configuration overheads. Dyn waits for the GIL at \( P_{\text{slow}} \), then runs at \( P_{\text{turbo}} \) while holding the GIL and switches to \( P_{\text{base}} \) after releasing it. While the workloads Pi and Regex do not scale, Dyn supports at least the execution at \( P_{\text{turbo}} \). The performance and power implications are in line with our synthetic benchmark results (Section 6.5.3) and the cost model (python in Table 6.6 greater than \( t_{\text{CS}} \) in Section 6.5.4).

For the workloads bz2 and SHA1, the performance benefit reaches its maximum at 4 threads because we pin the threads such that each runs on a different module, giving the thread full P-state control. When two threads run on a module, more P-state transitions are required per
package that eliminate the performance benefit at 8 threads. Own runs all threads at $P_{\text{base}}$ and boosts temporarily to $P_{\text{HW}}^1$ while holding the GIL. This manifests in a higher throughput when the GIL is held for long periods but for $\text{bz2}$ and $\text{SHA}$ the cost of requesting a P-state transition is not amortized by the higher frequency. Wait runs at $P_{\text{turbo}}$ if permitted by the TDP and only switches to $P_{\text{slow}}$ while waiting for the GIL. This strategy works well with high contention but introduces significant cost if the waiting period is too short (see Table 6.6).

In Figure 6.7 we compare Intel’s results for boosting disabled ($\text{Base}$) and enabled automatically by the processor ($\text{Auto}$). Overall, the results are similar to the ones obtained on AMD and what we expect from Section 6.5.2: The level of boosting depends on the number of halted cores, which enables $P_{\text{turbo}}$ for $\pi$ and $\text{Regex}$. $\text{SHA1}$ and $\text{bz2}$ boost slightly because not all processor features are used. The performance drop beyond 4 threads is due to hyper-threading.

### 6.6.2 SOFTWARE TRANSACTIONAL MEMORY

FASTLANE [Wam+13a] is a software transactional memory (STM) implementation that processes a workload asymmetrically (see Chapter 5). The key idea is to combine a single fast master thread that can never abort with speculative helper threads that can only commit if they are not in conflict. The master thread has only a very lightweight instrumentation and runs close to the speed of an uninstrumented sequential execution. To allow helper threads to detect conflicts, the master thread must make the in-place updates of its transactions visible (by writing information in the transaction metadata). The helpers perform updates in a write-log and commit their changes after a validation at the end of the transaction. The benefit is a better performance for low thread counts compared to other state-of-the-art STM implementations (e.g., TINYSTM [FFR08]) that typically suffer from the high instrumentation and bookkeeping overhead.

We used integer sets that are implemented as a red-black tree (RB), a linked list (LL), a skip list (SL), or a hash set (HS) and perform random queries and updates [FFR08]. The parameters are the working set size and the update ratio. Either all threads run at $P_{\text{base}}$ ($\text{FL}$) or the master statically runs at $P_{\text{turbo}}$ ($\text{FL-BM}$) and the helpers at $P_{\text{slow}}$, except for the helper that runs on the same module as the master. Note that the master thread is determined dynamically. Moreover, we compare with TINYSTM ($\text{Tiny}$) and uninstrumented sequential execution ($\text{Seq}$) at $P_{\text{base}}$. Our evaluation on the AMD processor shows in Figure 6.8 that running the master and helpers at different speeds ($\text{FL-BM}$) enables high performance gains compared to running...
all threads at $P_{\text{base}}$ (FL). The higher throughput can outweigh the higher power (50% vs. 2% for LL), thus, being more energy efficient. Tiny wins per design for larger thread counts. Table 6.7 shows that the master can asymmetrically process more transactions at $P_{\text{turbo}}$. While the helpers at $P_{\text{slow}}$ can have more conflicts caused by the master, the conflict rate caused by other slow helpers does not change. Dynamically boosting the commits of the helpers did not show good results because the duration is too short.

We chose this workload to highlight the importance of making P-state configuration accessible from the user space. It allows developers to expose properties of the application that would otherwise not be available to the processor. For applications that contain larger amounts of non-transactional code, supporting the ability to remotely set P-states for other cores would be very helpful. When a master transaction is executed, it could slow down the other threads in order to get fully boosted for a short period.

### 6.6.3 Hash Table Resize in Memcached

Memcached is a high performance caching system based on a giant hash table. While for the normal operation a fine-grained locking scheme is used, the implementation switches to a single global spinlock that protects all accesses to the hash table during the period of a resizing. The resize is done by a separate maintenance thread that moves items from the old to the new hash table and processes a configurable number of buckets per iteration. Each iteration acquires the global lock and moves the items in isolation.

Our evaluation was conducted with Memcached version 1.4.15 and the mc-crusher workload generator. We used the default configuration with 4 worker threads that we pinned on 2 modules. The maintenance thread and mc-crusher run on their own modules. The workload
Table 6.8: Memcached hash table resize statistics.

<table>
<thead>
<tr>
<th>Bulk Move</th>
<th>Strategy</th>
<th>Ops/s</th>
<th>Resize 10MB</th>
<th>Resize 1280MB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ms</td>
<td>stalled</td>
</tr>
<tr>
<td>10k</td>
<td>baseline</td>
<td>535k</td>
<td>96</td>
<td>63%</td>
</tr>
<tr>
<td>10k</td>
<td>stat resizer</td>
<td>547k</td>
<td>15</td>
<td>82%</td>
</tr>
<tr>
<td>10k</td>
<td>dyn resizer</td>
<td>547k</td>
<td>15</td>
<td>81%</td>
</tr>
<tr>
<td>10k</td>
<td>dyn worker</td>
<td>535k</td>
<td>18</td>
<td>82%</td>
</tr>
<tr>
<td>100</td>
<td>baseline</td>
<td>529k</td>
<td>24</td>
<td>66%</td>
</tr>
<tr>
<td>100</td>
<td>stat resizer</td>
<td>540k</td>
<td>22</td>
<td>86%</td>
</tr>
<tr>
<td>100</td>
<td>dyn resizer</td>
<td>508k</td>
<td>30</td>
<td>56%</td>
</tr>
<tr>
<td>100</td>
<td>dyn worker</td>
<td>461k</td>
<td>48</td>
<td>60%</td>
</tr>
<tr>
<td>1</td>
<td>baseline</td>
<td>237k</td>
<td>770</td>
<td>72%</td>
</tr>
<tr>
<td>1</td>
<td>stat resizer</td>
<td>245k</td>
<td>721</td>
<td>94%</td>
</tr>
<tr>
<td>1</td>
<td>dyn resizer</td>
<td>209k</td>
<td>893</td>
<td>62%</td>
</tr>
<tr>
<td>1</td>
<td>dyn worker</td>
<td>90k</td>
<td>1886</td>
<td>64%</td>
</tr>
</tbody>
</table>

The Turbo Diaries: Application-controlled Frequency Scaling Explained

The generator sends a specified number of set operations with distinct keys to Memcached, which result in a lookup and insert on the hash table that will eventually trigger several resizes. The hash table is resized when it reaches a size of $2^n \times 10\text{MB}$. The cache is initially empty and we insert objects until the 7th resize of $2^n \times 10\text{MB}$ ($1280\text{MB}$) is finished.

For the intervals in which the maintenance thread was active, we gathered for the first (10MB) and the last (1280MB) resize interval. These are reported in Table 6.8: number of items that are moved during one iteration (bulk move, configurable), rate of set operations during the entire experiment (ops/s), length of the resize interval (ms), the number of (stalled) instructions and average frequency achieved by the maintenance thread (freq).

We applied the following strategies during the resizing period: baseline runs all threads at $P_{\text{base}}$, stat resizer runs the maintenance thread at $P_{\text{turbo}}$ for the entire period, dyn resizer switches to $P_{\text{turbo}}$ only for the length of an bulk move iteration and causes additional transition overheads, dyn worker switches to $P_{\text{slow}}$ while waiting for the maintenance thread’s iteration to finish. The last strategy does not show a performance improvement because the cost cannot be amortized especially when the bulk move size gets smaller. The stat resizer shows the best performance because it reduces the resizing duration.

While the benchmark shows the benefit of assigning heterogeneous frequencies, an interesting observation is that the speedup achieved by boosting is limited because the workload is mainly memory-bound. Compared to baseline, stat resizer shows only a speedup of the resize interval between 7%–9% while it runs at a 22% higher frequency. The higher the frequency, the more instructions get stalled due to cache misses that result from the large working set. The number of stalled instructions effectively limit the number of instructions that can be executed faster at a higher frequency. On the other hand, the high cost of the P-state transitions in the dynamic strategy dyn resizer is hidden by an decreased number of stalled instructions but it still cannot outweigh the transition latency. Memcached’s default configuration performs only a single move per iteration, which according to our results shows the worst overall duration of the experiment (ops/s). A better balance between worker latency and throughput is to set bulk move to 100. With this configuration, memcached spends 15% of its execution time for resizes, which we can boost by 10%. This reduces the total execution time by 1.5% and allows 1.5% more ops/s because the worker threads spent less time spinning. Combined, this amortizes the additional boosting energy.
6.6.4 DELEGATION OF CRITICAL SECTIONS

We have shown that critical sections (CS) need to be relatively large to outweigh the latencies of changing P-states. Remote core locking [Loz+12] (RCL) is used to dedicate a single processor core to execute all application’s CS locally. Instead of moving the lock token across the cores, the actual execution of the critical section is delegated to a designated server. We leverage this locality property by statically boosting the RCL server and eliminate the P-state transition overhead for small CS.

We experiment with three of the SPLASH-2 benchmarks [Woo+95] and the accompanying version of BerkeleyDB [OBS99].

We report the speedup for all workloads over the single-threaded baseline P-state in Figure 6.9, and find that we obtain only incremental performance gains for the boosted cases. We show various combinations of worker P-states (reported as “W Px”) and P-states for the RCL server core (“R Px”), and contrast these with configurations where all cores run at P_{base} (“All P2”) and P_{4HW} (“All P4”) for comparison. Note that we do show standard deviation of 30 trials, but there is hardly any noise visible. We do not reduce the P-state for the waiting workers (due to latency reasons), but it seems there is enough TDP headroom for the brief RCL invocations to run even at P_{1HW} and we get speedups of 4% - 9%. As expected, the relative boost is larger if we start from a lower baseline at P_{4HW}. Overall, scalability of the benchmarks is good, reserving one core exclusively for RCL will cap scalability at 7 (worker) threads. The authors of RCL claim, however, that reserving this single core pays off in comparison to cache coherence traffic arising from spinlock ownership migrating between cores.

Focusing our attention on the CS, we find them to be short (with a peak at $\sim$488ns) for the selected benchmarks. To better understand the cost of communication and its behavior under various boosting scenarios, we implemented the core of the RCL mechanism, simple cross-thread polling message passing with two threads, in a small micro-benchmark. We report results for select configurations in Table 6.9 for AMD, which reflect unloaded latency with no competition for communication channels. Overall we were surprised by the round-trip delay when crossing modules, 480ns, vs. 91ns when communicating inside a module (both at P_{base}). Intra-module communication benefits greatly from boosting (91ns vs. 70ns), due to both communication partners and the communication link (shared L2 cache) being boosted. Communicating cross-module, boosting has a smaller performance impact on the...
Table 6.9: Core to core memory transfer latency (ns) for an average round-trip (work iterations: \( N_{\text{Worker}} : N_{RCL} \), 0.65ns each).

<table>
<thead>
<tr>
<th>Config</th>
<th>All P2</th>
<th>WP 2RP 1</th>
<th>WP 4RP 1</th>
<th>All P4</th>
<th>WP 4RP 0</th>
<th>WP 2RP 0</th>
</tr>
</thead>
</table>

communication latency (480ns vs. 445ns, via L3 cache), which helps to explain the small benefit seen in our workloads with short CS.

### 6.7 SUMMARY

We presented a thorough analysis of low-level costs and characteristics of DVFS on recent AMD and Intel multicore processors and proposed a library, TURBO\(^3\), that provides convenient programmatic access to the core’s performance states. The current implementation by hardware and OS is optimized for transparent power savings and for boosting sequential bottlenecks. Our library allows developers to boost performance using properties available at application-level and gives broader control over DVFS. We studied several real-world applications for gains and limitations of automatic and manual DVFS. Manual control exposes asymmetric application characteristics that would be otherwise unavailable for a transparent optimization by the OS. Limitations arise from the communication to memory and other cores that restrict the IPC. Our techniques, while useful today, also bring insights for the design of future OS and hypervisor interfaces as well as hardware DVFS facilities.

For the future, we plan to add an automatic dynamic tuning mechanism: based on decorated thread control structures, e.g., locks, we can obtain profiling information and predict the optimal frequency for each core. We also envision use cases beyond optimizing synchronization, such as DVFS for flow-based programming with operator placement (deriving the frequency from the load factor) or data routing (basing DVFS on deadlines or priorities). Finally, the TURBO library provides a research testbed to simulate future heterogeneous multicore processors with fast/slow cores, as well as to evaluate algorithms targeting energy efficiency or undervolting.

\(^3\)https://bitbucket.org/donjonsn/turbo
7 CONCLUSION AND FUTURE WORK
Software development will be confronted in the future with major challenges that result from required design shifts of the underlying hardware. Dennard scaling is considered broken because the energy improvements did not scale with the increasing frequency. However, Moore's law remains valid and provides an increasing number of transistors to fit on a processors chip. After the shift from ever increasing clock speeds towards multicore scaling, hardware development must introduce new means to improve the energy efficiency such that it can make an optimal use of the transistors. It is predicted that the next shift will be towards asymmetric and heterogeneous cores that one cannot afford to power simultaneously. Instead, depending on the workload characteristics a large number of smaller cores will speed up the execution using extensive parallelism while a small number of larger cores serve as accelerators for code that cannot be parallelized or that runs more efficiently on specialized cores dedicated to certain classes of algorithms. At the same time, it is predicted that hardware will become less reliable due to increased sensitivity, process variations and ware-out over time.

Software must allow a speedup that scales with large numbers of cores, which requires for many algorithms an efficient synchronization. It must support the dynamic adaption to asymmetric and heterogeneous hardware, including a scheduling to dedicated cores and auto-tuning of algorithms. Software will also gain more control over the hardware such that it can be configured according to the workload characteristics at runtime. Besides improving the performance, software is also required to tolerate hardware faults: Error correction enables threads to survive single event upsets and a robust synchronization ensures progress if the communication among threads is hampered.

This chapter summarizes the main achievements of this thesis in the field of resilience and efficiency of parallel computing and provides a brief outlook of possible future research directions.

7.1 Conclusion

Our contributions are twofold: We investigate means (1) to make software resilient to hardware faults, and (2) to improve the efficiency of parallel software. Based on the predictions of future hardware developments, we derived challenges towards software. These challenges require the software to become resilient against hardware faults and increase the efficiency and performance of parallel execution. While we cannot provide a "one size fits all" solution, we provide a number of algorithms and tools that help software to stand in the era of asymmetric processing on unreliable hardware.

We presented a universal TM construction based on a wait-free STM algorithm and showed that it guarantees maximal non-blocking progress under the assumptions of the the asynchronous multicore system model. It isolates the execution of transactions in order to protect the application from crash and non-termination failures. The processing of all correct transactions is assured as long as one thread survives because it will help processing transactions that were originally scheduled onto other threads. Since the upper bound of steps a transaction takes is unknown in asynchronous systems, we assign a monotonically increasing number of steps to the transaction for processing and abort the transaction if it exceeds its budget. The number of steps will eventually grow large enough to execute any correct transaction.

Based on the assumptions and isolation techniques of the universal TM construction, we derived a practical robust STM implementation, called R0BUSSTM. It is built on the multicore system model, which captures the properties of modern processors and operating systems. For efficiency reasons, we use a lock-based design that achieves blocking maximal progress. When progress is at stake, we first make reads visible to other threads to overcome repeated aborts due to read-write conflicts. If transactions still have problems to reach their commit, they enter an exclusive priority mode that allows a safe lock stealing from conflicting transac-
Conclusion and Future Work

The lock stealing prevents crashed and non-terminating transactions that still hold locks from blocking correct transactions. We must deal with crashed and non-terminating transactions explicitly because they also can enter the priority mode. Therefore, the thread in priority mode is inspected by all other threads and gets only a limited number of steps to execute prioritized. If it exceeds its budget, the execution is isolated in a separate process to check if the transaction can indeed reach its commit and how many steps it takes. Overall, ROBUSTM's performance in the good case is comparable to state-of-the-art STM implementations.

The above approaches deal with symptoms caused by hardware faults and software bugs such that their impact does not prevent correct threads and transactions from making progress. Transactional encoding is a fault tolerance mechanism that enables a thread itself to mask transient hardware errors. Arithmetic codes add redundancy to the data and provide an end-to-end error detection mechanism of symptoms such as bit-flips and erroneous arithmetic operations. The code is executed within transactions, which is used as a checkpointing technique for failure atomicity and performs a transparent replication of all updates to shared state. If the backward recovery by aborting and re-executing the transaction was not successful, we use the replicated state to repair the global state from the last update of the memory location. Transactional encoding is capable of converting the vast majority of errors detected by AN encoding into correct executions. Furthermore, it can correct some errors that were not detected by AN encoding.

We addressed the efficiency of synchronization by proposing FASTLANE, a lightweight STM algorithm that is optimized for smaller sets of threads. Using compiler generated code paths, it can dynamically adapt to the available hardware resources by selecting an appropriate synchronization scheme. In FASTLANE mode, the master thread has only minimal overhead and executes almost at the speed of the sequential uninstrumented code path. The speculative helper threads enable the speedup of the workload. A single helper is sufficient in most cases to outperform the sequential execution. The asymmetric processing of transactions gives FASTLANE a head start compared to state-of-the-art STM implementations, which we outperform for sets with few threads on most benchmarks. The master runs pessimistically, i.e., it never aborts, and guarantees progress. Helper threads can request to switch to the master code path if they encounter problems to reach their commit. Workloads with data partitions can benefit from multiple master threads concurrently. We introduced the communities of interest benchmark and showed that streamlining the synchronization overhead with the master for each partition can improve the performance significantly.

Finally, we presented an in-depth study of asymmetric cores in current hardware that are exposed by DVFS. Based on the investigation of the boosting features that allow cores to run above the base operating frequency, we developed the TURBO library that enables software to control the hardware. The library supports the setup and configuration of DVFS and a profiling of the performance inside applications. From the measured performance state transition latencies, we derived a cost model that allows to estimate when boosting pays off. Combined with the transition latencies for power gating, we concluded when blocking is to prefer over spinning in lock implementations. We proposed several strategies to initiate frequency transitions and reduce the latency cost for the threads. The evaluation of user-level DVFS control showed performance gains using several real-world workloads. The gains result from application knowledge that is exposed to assign heterogeneous frequencies.

7.2 FUTURE WORK

The upcoming shift towards heterogeneous processors and unreliable hardware unfolds a large field of research challenges. One crucial aspect is to provide software developers algorithms and tools to deal with these hardware challenges. We provide a set of such algorithms and tools in this thesis that can be used in combination or as a base to conduct future research.
Besides the follow-up work suggested in each chapter, there are several options that future work should investigate.

The combination of ROBUSTM and transactional encoding would provide multi-threaded applications with a robust synchronization and tolerance of transient hardware errors. That way, applications can rely both on progress guarantees and toleration of hardware faults. The major difficulty is to apply encoding techniques to multi-threaded programs because of the underlying arithmetic codes. Transactional memory can provide a wrapper for the underlying memory with multi-version support for each thread and mediation between different codes. Since the overhead of AN encoding in non-negligible, a combination with alternative error detection mechanisms, which are possibly implemented in hardware, can improve the performance significantly.

FASTLANE shows performance benefits using asymmetric instrumentation of code that can be further facilitated by asymmetric frequencies of cores. A hardware implementation would have to overcome the best-effort guarantees of current HTM implementations to allow the master thread to always commit. The solution would include multiple transactional modes, one speculative best-effort mode and as an extension a pessimistic mode that only reflects its performed updates but always wins conflicts. Future heterogeneous multicore processors can implement a transactional mode depending on their purpose, e.g., the large number of small parallel cores supports only the speculative mode while the large accelerating cores can execute pessimistically with low overhead.

The TURBO library can be used as a testbed for either heterogeneous cores by assigning asynchronous frequencies or as a testbed for dependable software by setting the cores to an unstable state. For a heterogeneity testbed, an extended support for DVFS control of memory, GPU/APU combinations and support for more platforms such as ARM would be beneficial. As a result, the interface for application-level DVFS control can be further generalized. For a dependability testbed, the operating system must run isolated on stable cores to allow a continuous operation and observation of the test application running on unstable cores, which are configured to be undervolted [VWF14]. A study of which fault symptoms or state corruptions become visible on application-level allows to validate and contribute to well-accepted fault models. Furthermore, it is worth investigating the reproducibility and variability of the observed faults to allow a configuration of the testbed.

A problem left open by this work is the dynamic tuning of the different proposed solutions. Heterogeneous computing imposes great challenges on the adaption of algorithms because predetermined static optimizations may not apply to the many possible processor configurations or variants. Instead, tuning will be even more required to be performed dynamically at runtime. The large variety of tuning options ranges from an efficient distribution of the workload to the heterogeneous cores and the configuration of the cores to the selection of the synchronization strategy for multi-threaded applications, which depends on the efficiency for performance and robustness for progress guarantees. The input for the tuning parameters can result explicitly from innovative programming models that are heterogeneity-aware (e.g., with annotations) or transparently from the profiling facilities of the runtime.
BIBLIOGRAPHY


Bibliography


LISTS OF FIGURES, TABLES AND ALGORITHMS
**List of Figures**

1.1 Overview of hardware developments and its challenges ............................................... 3
1.2 Scaling of transistors, frequency and cores ................................................................. 5
1.3 Scaling of power, voltage and performance ................................................................. 5
1.4 Scaling of cache levels and sizes ......................................................................................... 6
1.5 Scaling of Amdahl’s law ..................................................................................................... 8
1.6 Overview of progress conditions ....................................................................................... 14
1.7 Overview of approaches towards software challenges ....................................................... 18

2.1 Overview of the Asynchronous Multicore System Model .................................................. 28
2.2 State representation in the universal TM construction ..................................................... 31
2.3 Assignment of invocations to threads for processing ....................................................... 31
2.4 Dealing with thread and transaction crashes ................................................................... 35

3.1 Data structures for the lock-based design of TinySTM ...................................................... 46
3.2 Illustration of locks used in RobustSTM .......................................................................... 47
3.3 States during the lifetime of a RobustSTM transaction .................................................... 48
3.4 Dealing with non-terminating transactions in RobustSTM ............................................. 50
3.5 Comparison of lock-based vs. non-blocking STM ......................................................... 51
3.6 Performance of RobustSTM with micro-benchmarks and STAMP applications .......... 51
3.7 Performance with injected crashes and non-terminating transactions ............................. 52
3.8 Throughput over time intervals with non-terminating transactions present ................. 53

4.1 Overview of transactional encoding .................................................................................... 60
4.2 Code word domain of arithmetic codes and possible computation [Sch11] ................... 61
4.3 Overview of the write-set .................................................................................................. 63
4.4 Overview of the XOR extension ....................................................................................... 66
4.5 Evaluation of application behavior with injected errors ................................................... 69
4.6 Slowdown compared to the native execution of applications .......................................... 70
4.7 Cost of error tolerance for different error rates ............................................................... 71

5.1 Bridging the gap between sequential and STM performance at low thread counts .......... 76
5.2 Overview of shared variables used by Fastlane algorithms ............................................. 79
5.3 Overview of generated code paths ................................................................................... 80
5.4 Throughput of the Intset benchmarks ............................................................................ 86
5.5 Completion times of the STAMP benchmarks ................................................................. 87
5.6 Merging of windows into the communities of interest ..................................................... 91
5.7 Throughput of the COI benchmark .................................................................................. 92

6.1 Organization of an AMD FX-8120 processor .................................................................... 100
6.2 Overview of Turbo library components .......................................................................... 102
6.3 Characteristics of blocking and spinning ........................................................................ 107
6.4 Frequency sequence for spinning, blocking, scaling and critical regions ....................... 108
6.5 Characteristics of manual P-state control ...................................................................... 109
6.6 ccbench throughput (AMD FX-8120) ............................................................................. 111
6.7 ccbench throughput (Intel i7-4770) ............................................................................... 112
6.8 Fastlane STM integer set benchmarks .......................................................................... 113
6.9 Throughput of SPLASH-2 and BerkeleyDB .................................................................... 115
# LIST OF TABLES

1.1 Overview of contributions ................................................. 20
4.1 Overview of tolerated state and write-set corruptions .................. 66
4.2 Combinations of the transactional encoding extensions .................... 67
4.3 Ratio of tolerated transient errors ...................................... 68
4.4 Comparison of the write-set size ....................................... 68
5.1 Shared variables used by FASTLANE algorithms ........................... 78
5.2 STMs and FASTLANE configurations used in the tests ...................... 87
5.3 Contribution of helpers to the global commit throughput .................. 89
5.4 Impact of optimizations for FL-O3 ...................................... 90
5.5 Contribution of helpers to the global commit throughput .................. 92
5.6 Abort rates for STMs and FASTLANE .................................... 93
6.1 Specification of the AMD and Intel processors ........................... 99
6.2 Default P-state configuration of AMD FX-8120 .......................... 100
6.3 Default P-state configuration of Intel i7-4770 .......................... 101
6.4 Latency cost (AMD FX-8120) ........................................ 105
6.5 Latency cost (Intel i7-4770) ........................................... 106
6.6 ccbench characteristics .................................................. 111
6.7 Commit ratio of the total commits by the FASTLANE master .............. 113
6.8 Memcached hash table resize statistics ................................... 114
6.9 Core to core memory transfer latency .................................... 116
LIST OF ALGORITHMS

1.1 Example of a memory transaction ........................................ 16
2.1 Thread functions for the processing of invocations .................. 32
2.2 Universal TM algorithm .................................................. 34
3.1 CAS loop that might not terminate .................................... 44
3.2 Terminating CAS loop with atomic-or .................................... 44
4.1 Counter applications ..................................................... 64
4.2 TM for dependability ...................................................... 65
5.1 Code path selection at transaction start .............................. 80
5.2 Master code path ......................................................... 81
5.3 Helper code path ........................................................... 82