EFFICIENT RUNTIME SYSTEMS FOR SPECULATIVE PARALLELIZATION

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by Clemens Hammacher

Saarbrücken
February 2017

UNIVERSITÄT
DES
SAARLANDES

Day of Colloquium	27.07.2017
Dean of the Faculty	Univ.-Prof. Dr. Frank-Olaf Schreyer
Chair of the Committee	Prof. Dr. Jan Reineke
Reporters:	
First Reviewer	Prof. Dr. Andreas Zeller
Second Reviewer	Prof. Dr. Sebastian Hack
Third Reviewer	Dr. Björn Franke
Academic Assistant	Dr. María Gómez Lacruz

SAARLAND UNIVERSITY

Zusammenfassung

Chair of Software Engineering and Compiler Design Lab
Department of Computer Science

Dissertation

Efficient Runtime Systems for Speculative Parallelization

by Clemens Hammacher

Manuelle Parallelisierung ist zeitaufwändig und fehleranfällig. Automatische Parallelisierung andererseits findet häufig nur einen Bruchteil der verfügbaren Parallelität. Mithilfe von Spekulation kann jedoch auch für komplexere Programme ein Großteil der Parallelität ausgenutzt werden. Spekulativ parallelisierte Programme benötigen zur Ausführung immer ein Laufzeitsystem, um die spekulativen Annahmen abzusichern und für den Fall des Nichtzutreffens die korrekte Ausführungssemantik sicherzustellen. Solche Laufzeitsysteme sollen die Ausführungszeit des parallelen Programms so wenig wie möglich beeinflussen. In dieser Arbeit untersuchen wir, inwiefern aktuelle Systeme, die Speicherzugriffe *explizit und in Software* beobachten, diese Anforderung erfüllen, und stellen Änderungen vor, die die Laufzeit massiv verbessern. Außerdem entwerfen wir zwei neue Systeme, die mithilfe von *virtueller Speicherverwaltung* das Programm indirekt beobachten und dadurch eine deutlich geringere Auswirkung auf die Laufzeit haben. Eines der vorgestellten Systeme ist mittels eines *Moduls* direkt in den Linux-Betriebssystemkern integriert und bietet

so die bestmögliche Effizienz. Darüber hinaus bietet es weitreichendere Sicherheitsgarantien als alle bisherigen Techniken, indem sogar Systemaufrufe zum Beispiel zur Datei Ein- und Ausgabe in der spekulativen Isolation mit eingeschlossen sind. Wir zeigen an einer Reihe von Benchmarks die Überlegenheit unserer Spekulationssyteme über den derzeitigen Stand der Technik. Sämtliche unserer Erweiterungen und Neuentwicklungen stehen als *open source* zur freien Verfügung.

Diese Arbeit ist in englischer Sprache verfasst.

SAARLAND UNIVERSITY

Abstract

Chair of Software Engineering and Compiler Design Lab
Department of Computer Science

Dissertation

Efficient Runtime Systems for Speculative Parallelization

by Clemens Hammacher

Manual parallelization is time consuming and error-prone. Automatic parallelization on the other hand is often unable to extract substantial parallelism. Using speculation, however, most of the parallelism can be exploited even of complex programs. Speculatively parallelized programs always need a runtime system during execution in order to ensure the validity of the speculative assumptions, and to ensure the correct semantics even in the case of misspeculation. These runtime systems should influence the execution time of the parallel program as little as possible. In this thesis, we investigate to which extend state-of-the-art systems which track memory accesses *explicitly in software* fulfill this requirement. We describe and implement changes which improve their performance substantially. We also design two new systems utilizing *virtual memory abstraction* to track memory changed implicitly, thus causing less overhead during execution. One of the new systems is integrated into the Linux kernel as a *kernel module*, providing the best possible performance. Furthermore it provides stronger soundness guarantees than any state-of-the-art

system by also capturing system calls, hence including for example file I/O into speculative isolation. In a number of benchmarks we show the performance improvements of our virtual memory based systems over the state of the art. All our extensions and newly developed speculation systems are made available as *open source*.

To Anna, with love.

Acknowledgements

This thesis would not have been possible without substantial support from many people.

First and foremost, I want to thank my advisors Andreas and Sebastian, who both saw my potential before I saw it myself, and put a lot of trust in me right from the beginning. Thank you, Andreas, for arranging funding for my research and for providing such a great working environment. I enjoyed the confidence you put in all your PhD students, which gave me ample freedom to pursue my research. Thank you, Sebastian, for all the time you invested already as a Junior Professor to get down to the bottom of technical challenges. I always admired your steady urge to be directly involved in the work of your students, even after your responsibilities and your group grew greatly after you assumed your full professorship.

I am also thankful for all the discussions with my collegues, be it at the retreats in Dagstuhl or Otzenhausen, or individually at the university. Especially in tough times of setbacks and failure, the support from people who knew the struggle and overcame it was invaluable. Special thanks go to Kevin, my good friend and collegue from my first semester onwards. We shared an office for more than five years and went through numerous ups and downs together. Thank you for always looking forward and keeping our team in motion!

I am deeply grateful for all the support I received from my parents throughout my life, putting me in the position to successfully complete this dissertation. Thank you for all the effort it took you to get me to where I am today.

And last but not least I want to thank Anna for her love, her patience and her endless support throughout the last years. I would not have gotten to know you without my postgraduate studies for my doctorate, and this thesis would not exist without you.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The increase in the density of integrated circuits continues till today. Since the beginning of the millennium, however, processor manufacturers struggle to translate this to increased clock speeds, and thus increased single-thread performance. The limiting factor is mostly *power consumption*: It increases exponentially with a linear increase of frequency. Figure 1.1 shows that around the year 2000, power consumption reached the critical level of 100 watts. Dissipating the produced heat becomes increasingly difficult if this level is exceeded, making such processor designs uneconomical. However, Figure 1.1 also shows that the number of transistors continues to grow exponentially. This is because the size of the semiconductors—constituting the transistors—can still be reduced, allowing Moore's law to hold true at least for the next couple of years[1]. These additional transistors are used by processor vendors to place *multiple processor cores* on a single die. From 2005 onwards we see an exponential growth of the

[1] The current manufacturing process uses 14 nm structures, with 10 nm technology being developed. At about 5 nm, the structures cannot be reduced any further on silicon based dies due to increasing quantum tunneling effects.

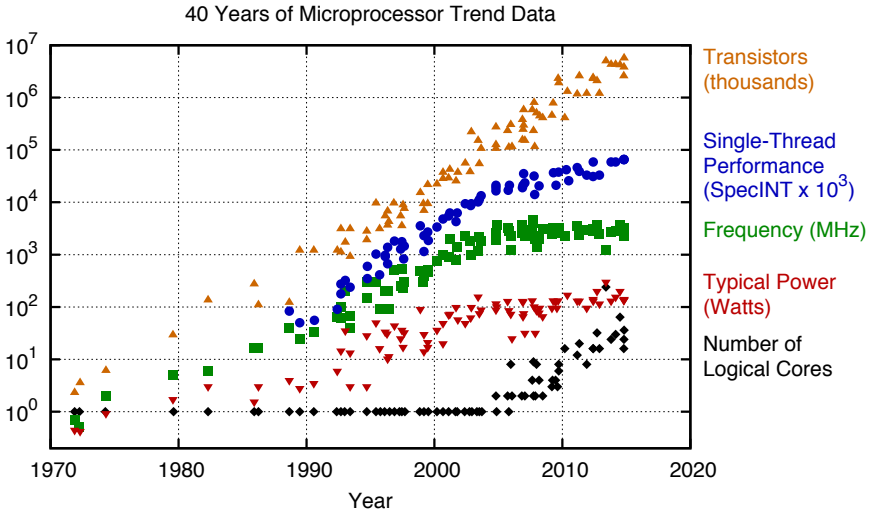


FIGURE 1.1: The number of transistors is growing exponentially over the full time range, while frequency and power consumption stopped increasing shortly after the year 2000. Instead, the number of cores starts rising exponentially from that time on. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for 2010–2015 by K. Rupp [93].

number of logical cores in a processor. This development, however, shifts the burden to translate advances on the processor's side into increased software performance to the software developers.

In order to benefit from the computational power of multiple cores, software needs to be *parallelized*. The traditional approach is manual parallelization. It requires expert programmers that fully comprehend the dependencies within the software that should be parallelized, and are able to introduce the right set of synchronization mechanisms to still guarantee correctness of the program while exposing the maximum amount of parallelism. As manual parallelization is a

tedious and error-prone task, and the resulting program is fragile with respect to later changes of the code, it is only done for specific and important software. Nowadays the majority of software still executes single-threaded.

The alternative to manual parallelization is automatic parallelization. It does not require the skill set and investment of time of manual parallelization. Instead, the traditionally developed sequential program is analyzed and automatically transformed into a parallel program. Over the last decades, many approaches have been presented which are able to handle different classes of programs. All these parallelization schemes, however, necessitate precise information about dependences in the code. Such dependence analysis is successful for computation kernels with regular memory access patterns, as they often occur in scientific or mathematical computations. If programs get bigger, the picture changes. Dynamic data structures built around pointers often pose hard to solve problems for static analyses. In those cases, dependence analyses typically overapproximate and detect a potential dependence. As modern programs use many of such dynamic data structures, traditional automatic parallelization approaches fail to find a substantial amount of parallelism there.

Speculation is a technique that allows to parallelize such programs anyway. By making optimistic assumptions—e.g. assuming that a potential dependence will not manifest at runtime—parallel code can be generated. The execution of speculatively parallelized code is not guaranteed to succeed at runtime, however. If one of the optimistic assumptions does not hold, the program might produce wrong output or might even crash. Hence, a runtime system is needed to guard the execution against such *misspeculations*, and bring it

back to a safe and correct execution state. This thesis investigates and improves the state of the art in software-only runtime systems for speculative parallelization, and argues that such systems are often insufficient for automatic speculative parallelization. We then present novel approaches that use facilities of the operating system and the underlying hardware, and show that using these systems, automatic speculative parallelization often provides great speedups. Our main contributions are as follows:

- We apply *implicit memory tracking* in the form of *software transactional memory (STM)* to the problem of automatic speculative parallelization. We identify the main sources of overhead and propose and implement different solutions. We show that these changes reduce the overhead by orders of magnitude.

- We describe a *virtual-memory based* runtime system for speculative execution along the lines of previously published approaches. We evaluate its performance on several real-world programs and demonstrate an enormous performance benefit compared to STM.

- We further improve both performance as well as the isolation guarantees of the virtual-memory based system by implementing the main functionality directly in the Linux kernel. This is the first work describing such an implementation. In the evaluation, we demonstrate a further significant performance gain.

- We describe a novel approach to address the problem of *coarse granularity* of virtual-memory based systems. By instrumenting the program and keeping minimal metadata about the memory operation of the program, the granularity can be chosen

arbitrarily down to individual bytes. We show that this can be done with moderate overhead.

- We make all systems developed in this thesis available as open source.

1.1 Thesis Organization

The thesis is organized as follows:

- After introduction and clarification of general terms in Chapter 1, we review the state of the art in automatic parallelization in general, speculative parallelization, as well as runtime systems for speculative execution in Chapter 2.
- In Chapter 3, we introduce the Sambamba framework we developed to integrate the different approaches developed in this thesis and automatic parallelization approaches by my colleague Kevin Streit.
- Chapter 4 applies the state of the art software transactional memory system TinySTM to the problem of speculative parallelization, and shows how to improve the performance substantially.
- In Chapter 5 we introduce our U-TLS system, which implements a virtual-memory based runtime system for speculative parallelization in user space.
- Chapter 6 shows how to transfer the same concepts to the kernel space, and evaluates the performance gain of this system called

K-TLS. In Section 6.2, we supplement K-TLS with instrumentation for variable granularities and evaluate the performance impact of this addition. We call this system K-TLS$^+$.

- Finally, Section 7 concludes this thesis and lists ideas for future work.

1.2 Terminology

As different authors in the literature use different names for the same concepts, and sometimes mean different concepts by a specific term, we clarify the vocabulary used in this thesis in the following compilation of general terms related to speculative parallelization.

Parallelization. Parallelizing a piece of software means preparing it for parallel execution. There is a variety of approaches for parallelization. It can either be done statically (at or before compile-time), or dynamically (at run-time). In either case it can optionally be speculative. All these cases are further detailed below.

Static Parallelization. Parallelization is called static if it happens *before actually executing* the program. It can either be performed manually by a developer (potentially assisted by language extensions, libraries or compiler hints), or by a compiler which automatically determines appropriate code transformations and performs them at some stage during compilation. Also in static parallelization, there might exist different parallel variants of the same code, or also a sequential variant. If

just the decision which of these variants to execute is made at run-time, we still consider this static parallelization.

Static parallelization may also utilize *profiling information* generated at previous runs of the program.

Dynamic Parallelization. In dynamic parallelization, the *decisions where to parallelize* as well as the *generation of the parallel code* happen at run-time. This means that a *just in time (JIT)* compiler needs to be available to dynamically recompile parallelized functions. Typically, profiling information or other dynamic data is used to guide the dynamic parallelization decisions.

All dynamic parallelization approaches that we are aware of are *automatic* approaches.

Automatic Parallelization. As the name suggests, automatic parallelization is performed without involving interaction of a developer. Instead, static or dynamic *parallelization analyses*, typically consisting of points-to, alias or shape analyses, are used to find parallelizable locations in the program.

Speculative Parallelization. The parallelization of a specific code region is called *speculative*, if the soundness of its execution (with regards to the sequential semantics) cannot be inferred before actually starting the execution. This execution is then also called *speculative*. The sequential semantics might be violated with respect to the memory effects, i.e. the modifications to the virtual memory performed by the code, termination effects (non-termination or abortion), or other side effects like system calls executed, for example due to I/O effects.

In order to still provide soundness guarantees when executing speculatively parallelized code, runtime checks have to be installed to determine *misspeculations*.

Thread Level Speculation (TLS). Speculative execution is any execution that is performed without knowing whether the result can or will be used afterwards. Modern processors for example include *instruction level parallelism (ILP)*: They fetch and execute instructions in parallel or out-of-order in order to increase the throughput. By *speculatively* executing instructions— even memory instructions— without knowing yet whether they should really be executed, the amount if ILP can be drastically increased. This is mainly driven by *branch prediction*. Similarly, TLS is used to extract parallelism at the *thread level* as opposed to the instruction level by executing code either without knowing whether it would be executed in sequential execution, or by ignoring data dependences and thus potentially producing wrong results. Both of these techniques are described in the following.

Control Flow Speculation. This form of speculation assumes that certain branches will not be executed, hence it ignores all effects of these blocks. The decision where to speculate can be based on statistical execution frequencies, value profiles, or benefit driven (e.g. speculate that blocks containing calls to the `abort` function will not execute).

Detecting misspeculations of control flow speculation is cheap and does not require collecting any further data. Whenever the corresponding branch is taken, a misspeculation has happened.

Memory Speculation. This is the kind of speculation we mostly focus on. Memory speculation means to ignore possible memory effects of parallelized code or to assume non-aliasing of parallel memory accesses without being able to prove this beforehand. The goal is to reduce the number of data dependencies between speculative tasks.

Memory speculation always requires a sophisticated runtime system which tracks the memory regions accessed within parallel tasks, checks for overlaps and performs appropriate actions to recover from misspeculation.

Memory Conflict. A memory conflict (also called memory violation) is reported by the runtime system if it determines that the memory state produced by speculatively parallelized code might not be correct, thus it is a special case of *misspeculation*. In most cases, these checks are not precise, hence memory conflict must be *overapproximated*, leading to *false conflicts* being reported.

A memory conflict generally occurs if a speculative task has read a value from memory which was subsequently overwritten by another task *which commits first* (a minor relaxation of the Bernstein condition [5]). The detection of such conflicts can happen *eagerly* during transactional read or write operations, *lazily* (or *delayed*) during transactional commit, or concurrently by another processing unit.

False Sharing / False Conflict. In memory speculation systems (see above), most often memory is not tracked at the granularity of individual bytes, but in larger chunks. For TLS systems, this often is even the granularity of memory pages (4 kB on

most architectures). Hence disjoint *objects* in memory *share* the same metadata which tracks accesses to this memory, if they are located within the same *memory block* (defined by the granularity). Since accesses to those disjoint memory regions cannot be distinguished by the runtime system, memory conflicts have to be reported *pessimistically* whenever speculative tasks compete for the same block. If a finer granularity had resolved this *memory conflict*, we would call it *false conflict*, because another tracking scheme would not have reported it.

Runtime System. A runtime system is a software library which is available to the program under execution during its run-time, but does not belong to the program itself. It is often triggered by the executing program via *callbacks* (function calls into the runtime system) placed in the executed code, but it can also run concurrently to the executed program and interact with it proactively.

Examples of runtime systems are *just in time* compilers, *software transactional memory* or other speculation guarding systems.

Sequential Execution. Sequential (sometimes also called *serial*) execution is the execution of code on *one single thread*, hence it is the opposite of parallel execution. In some cases, also *parallelized code* will be executed sequentially, for example if speculative parallelization was detected to fail at run-time.

Speculative Task. A *speculative task* is the dynamic instantiation of one piece of speculative work. In the context of parallelization, speculative tasks are often optimistically executed in parallel

to each other, while a runtime system or the code itself checks for misspeculations. Each task will in the end either *commit* its changes, i.e. merge it into the non-speculative state, or *roll back* and re-execute either speculatively or non-speculatively.

Strong Atomicity vs. Weak Atomicity. A transactional memory guarantees that the effect of each transaction is either seen completely by other tasks or not at all. This concept is called *atomicity*. Also TLS systems want the individual tasks to execute *atomically*. There are two degrees of atomicity: A *strongly atomic* system guarantees atomicity with respect to all tasks, independent of whether they are using the same runtime system or not. Strong atomicity in general requires hardware support or special operating system support in order to prevent concurrent code from seeing partial updates during commit. Software TM or TLS solutions typically provide *weak atomicity* only, meaning that atomicity is only provided for other tasks using the same runtime system. If all memory operations for example are executed via the respective STM functions, then tasks observing inconsistent state are detected and re-executed, providing weak atomicity for all the committed tasks.

Single Global Lock (SGL) Semantics. This is the most simple semantics for executing multiple critical sections in parallel threads. The semantics is *as if* a single global lock would be taken before entering a section, and released then leaving it. This definition makes it very easy to reason about the semantics of a parallel program, as race conditions are excluded and the amount of nondeterminism is reduced significantly.

(Ir-)Regular Data Structure. Regular data structures are data structures with a well-defined shape in memory, like C-strings and arrays. As those structures occupy contiguous bytes in memory, they are typically easy to analyze statically.

Irregular data structures on the other hand are scattered over the memory space (often in the heap) and connected via pointers. Since pointers to different objects might be placed in the same memory, static analyses often have to overapproximate, making accesses to disjoint parts of the irregular data structure undistinguishable. Those data structures therefore cause problems for automatic parallelizers. If corresponding program locations are to be parallelized anyway, memory speculation is a resort.

1.3 Publications

This thesis builds on the following publications (in chronological order):

- **Thread-Level Speculation with Kernel Support.** In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, March 2016. Clemens Hammacher, Kevin Streit, Andreas Zeller, and Sebastian Hack.

- **Generalized Task Parallelism.** In *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 12, Number 1, January 2015. Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack.

- **Sambamba: Runtime Adaptive Parallel Execution.** In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems (ADAPT)*, January 2013. Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack.

- **SPolly: Speculative Optimizations in the Polyhedral Model.** In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 2013. Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack.

- **Sambamba: A Runtime System for Online Adaptive Parallelization.** In *Proceedings of the 21st International Conference on Compiler Construction (CC)*, March 2012. Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack.

- **Profiling Java Programs for Parallelism.** In *Proceedings of the ICSE Workshop on Multicore Software Engineering (IWMSE)*, 2009. Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller.

Chapter 2

State of the Art

In this chapter we investigate the state of the art in the field of speculative parallelization. To this end, we first review the most prominent and also recent automatic parallelization approaches, and then specifically focus on speculative parallelization and the runtime systems used during the execution of the resulting programs.

2.1 Automatic Parallelization

As there have been decades of research on parallelization, we focus on work which is relevant and related to the topic of this thesis. We thus exclude any languages or language extensions for manual parallelization, even if automatisms for extracting or enhancing parallelism were presented (like e.g. Galois [57] or PetaBricks [3]). As we focus on statically compiled languages, we also exclude previous work for languages which are executed in an interpreter or virtual machine (like e.g. Jrpm [18]).

In automatic parallelization, an essential step is determining the data dependencies and control dependencies within a program. In 1987, Ferrante et al. [33] came up with the notion of a *program dependence graph* (PDG) which encapsulates those dependencies and has since then often been used as the basis for parallelization and other program optimizations. If the nodes in the PDG carry information about the actual operation to be performed, it fully describes the semantics of a program, and can thus be used as an alternative representation.

Burke et al. [13] use the PDG to statically detect fork-join based parallelism. In contrast to other authors they not only focus on loops, but also detect parallelization opportunities in straight-line code. As their output is a source-code program again, they describe a Fortran-like target language featuring a DOALL construct for parallelizing loops and the COBEGIN and COEND keywords for marking parallel code sections. In order to remove some data dependencies they include a privatization analysis. Their parallelization algorithm works by first marking everything to be parallel, and then handling each data dependency by either privatizing the corresponding memory or serializing the respective tasks. They also consider low-level synchronization primitives to fulfill data dependencies between parallel tasks, but decide against them for performance and simplicity reasons.

Sarkar [101] describes a quite similar system which also statically generates structured parallelism from Fortran programs by partitioning the PDG. In contrast to Burke et al., Sarkar allows for data dependencies between parallel tasks and adds explicit synchronization via the WAITING keyword. He also discusses the trade-off between *ideal parallelism* exploiting all the parallelism in the program and *useful parallelism* excluding certain non-profitable opportunities. His

solution is the definition of a *cost function* containing overhead for task spawning and synchronization, and also includes profile data like loop frequencies and branch probabilities [102]. Based on this objective function, he iteratively merges parallel tasks if this decreases the predicted *parallel execution time*.

Saltz et al. [100] detect wavefront-parallelizable loops by using a combination of static and dynamic analyses. They statically detect so called *start-time schedulable* loops, for which the access pattern does not depend on any values computed inside the loop. Then they extract *inspector code* which assigns a wavefront number to each loop iteration, such that each iteration only has dependencies to iterations with smaller wavefront ids. They place code to execute the wavefronts in increasing order, and parallelize the inner loop, which executes all iterations within that wavefront. Overall, this builds a dynamic variant of *loop skewing*.

In 1994, the SUIF compiler infrastructure [123] is presented. It features its own intermediate representation (IR) and contains several analyses and transformations, including a loop parallelizer. It is based on an *array dependence analysis*, and optimizes for both parallelism and locality. For code generation, it translates back to C code, which is compiled by a traditional compiler and linked against a runtime library featuring parallel execution. Hall et al. [38] later extend this approach with a better parallelization analysis, including reduction and privatization detection.

Still in 1994, Rauchwerger and Padua publish the first *dynamic* and *speculative* parallelization approach, called the *privatizing DOALL test* [87]. They statically extract *inspector code* from loops which

determines whether there are any cross-iteration dependencies. If this is the case at runtime, the loop is executed sequentially. Otherwise, it is executed as a DOALL loop. They later extend this approach to also detect *partially* parallel loops [85]. Based on the information computed by the inspector code, the *scheduler* dynamically generates an execution schedule, which is then executed by the *executor code*. Their approach is also able to detect privatization and reduction opportunities during inspection. The applicability, however, is limited: In order to generate inspector code, the accessed memory locations must be known before entering the loop. They mainly focus on applications operating on one *shared array*, where only the subscripts of the array accesses need to be recorded.

One year later, they come up with an improved scheme known as the *LRPD test* [86]. It does not inspect the loop before executing it, but rather speculatively executes it as a DOALL loop and at the same time keeps track of the accessed memory locations (i.e. array subscripts). Afterwards it checks whether the loop was in fact fully parallel, and otherwise rolls back and re-executes sequentially. The problem of restoring the non-speculative state, however, is not addressed. They briefly discuss ideas how to solve this in software, but in the evaluation they instrument all programs manually.

Another important model, which can also be used for parallelization purposes, is the *polytope model* [29, 30, 76]. It describes a perfect loop nest of depth k as a convex k-dimensional polytope, where each integer point corresponds to one iteration of the innermost loop, and its position determines the values of the iteration variables of the surrounding loops. Also the dependencies between loop iterations are described by polyhedra. This allows to model most of the classical

loop transformations as affine transformations of these polyhedra. In fact, the problem of optimally *scheduling* the execution of a polyhedron subsumes most of these transformations. The scheduler often tries to minimize an objective function over the polyhedra. This way, also parallelization can be included in the output model of the scheduler, and can thus be seen as an optimization problem in the polytope model, as shown by Lengauer [60]. Beside generating maximum parallelization, the objective function can also include other factors like locality in the memory accesses, or it can be restricted by adding resource constraints. Also Feautrier explicitly describes automatic parallelization in the polytope model [28] and provides a nice summary of previous work. Others build on this work by also integrating loop splitting [22] or porting it to modern concepts like OpenMP-based parallelization [10].

Lim and Lam describe a similar system [62] which also maps iterations of a loop nest to a new domain using affine expressions, subsuming many existing loop transformations. In contrast to polyhedral techniques, they explicitly focus on parallelization, and they also support pipeline parallelism. Also, they do not overapproximate dependencies by dependence vectors.

In 1999, Rugina and Rinard [91] approach the problem of inaccuracies in state-of-the-art points-to analyses when it comes to pointer arithmetic and recursive algorithms. They generate a symbolic expression for each memory access, and propagate the expressions up to the method start. Using a fixed-point analysis, they recursively inline the expressions for called functions into the callee. They then analyze whether the memory accesses of adjacent function calls are disjoint, and parallelize independent function calls accordingly. The

evaluation demonstrates that this approach can be used for many classical divide-and-conquer algorithms like matrix multiplication or mergesort.

Rus et al. in 2003 describe a hybrid static and dynamic analysis to detect DOALL loops [95]: They statically construct the symbolic dependence set for a loop, and either prove it empty statically, or generate code to do the test dynamically when the actual input values are known. In 2007, they improve the approach by statically generating a disjunction of predicates which prove the dependence set empty [94], and then generate code to check the predicates at runtime. This improves the performance compared to the full emptiness check they were performing before.

Bhowmik and Franklin [6] build on the SUIF platform [38, 123] to create a *speculative parallelization framework*. They argue that speculation is absolutely necessary in order to extract larger amounts of thread-level parallelism from general-purpose applications. Using profile information, they statically find the best spawn point for each basic block and loop iteration. For the speculative execution they assume hardware support for lightweight thread spawning, detecting memory conflicts, and rollback. Even though they make optimistic assumptions about the performance of such hardware extensions, they also note that the *thread formation* is essential for guaranteeing success. After all, already in 1986 Sarkar and Hennessy showed that finding the optimal thread partitioning of a program is NP-complete [103].

Another important work is *decoupled software pipelining* (DSWP), as described by Rangan et al. in 2004 [84]. The original idea is a

hardware extension which can be used for (manual) parallelization. It allows to efficiently forward values from one thread to another in a FIFO manner, and is designed to efficiently implement pipeline parallelism. Parallel threads just issue *produce* and *consume* instructions annotated by an integer tag, where a *consume* returns the first unused produced value with the same tag, or blocks until such a value is produced by another thread. The proposed programming model would be to split a loop into several *pipeline stages* where each stage only has dependencies to previous stages or to itself. By explicitly forwarding the values for each dependence, the sequential semantic is preserved. Ottoni et al. [74] describe how to automatically generate such parallel programs using static analyses and transformations. Later extensions include support for multiple parallel instances of individual pipeline stages in order to compensate unbalanced stages and provide more parallelism [83]. Vachharajani et al. [117] add speculation support, but it only allows for control-flow speculation and assumes *versioned memory* for efficient rollbacks. Huang et al. [50] propose to parallelize individual pipeline stages using other independent parallelization techniques, and in other work propose to also parallelize iterations of different loops [49]. Even though no paper explicitly mentions this, later work like Parcae [82] suggests that DSWP now also works without special hardware support. Parcae itself, however, does not focus on parallelization, but on the automatic platform-wide tuning of parallelization in order to maximize overall throughput.

POSH [63], built on top of gcc, also assumes special hardware with support for thread level speculation (TLS). Using profiling data, it speculatively spawns loop iterations, loop continuations, function calls

or their continuation before reaching their actual sequential execution point. The assumption here is that the programmer gave enough structure to the program by encapsulating independent computations in subroutines or computing independent values in disjoint loop iterations. Thus, no loop transformations or preprocessing on basic block or instruction level is performed. If data dependencies are not fulfilled at the time when a speculative task is spawned, value prediction provides a probable input value.

Zhong et al. present another approach for speculative DOALL execution of loops [124]. They increase the amount of parallelism detected by identifying privatization and reduction opportunities and speculatively apply loop fission and other transformations. Also, they ignore long-distance cross-iteration data dependencies, as they are unlikely to cause rollbacks at runtime. During execution, they assume full hardware support for TLS, covering both the memory and register values. Additionally, they assume a hardware network to efficiently send register values between different cores.

In 2010, Vandierendonck et al. describe the Paralax [118] infrastructure, which aims to automatically parallelize irregular pointer- and control-intensive C applications. As static analyses are not good enough to precisely analyze such applications, they often have to fall back to overapproximations. Therefore, a set of annotations is proposed which can be used by the programmer to facilitate alias analysis or dependence analysis. Using the information provided via these annotations in combination with powerful analyses like the *data structure analysis* (DSA) [59], the authors are able to extract a substantial amount of pipeline parallelism from programs like `bzip2` that were previously considered as hard to parallelize.

Campanoni et al. provide an interesting new usage for sibling processors sharing the same first-level cache (as in *hyper-threading* on Intel architectures). Their HELIX [16] system is a classic loop parallelizer with explicit synchronization for control and data dependencies. They noticed that processors often stall while waiting for a cache line which was previously written by another core. This not only happens when forwarding values between parallel tasks, but also when acquiring a lock which was previously released by another core. In order to reduce these stalls, they use the second (virtual) core for prefetching values needed by the first core. This allows the first core to execute the actual code much faster since the chances for the accessed data to be present in the cache are increased. Later, they propose to add an explicit hardware mechanism for implementing locks more efficiently [15]. To that end, they propose an ISA extension with *signal* and *wait* instructions and a ring architecture which proactively forwards the state of locks to all other cores. This improves performance especially for short-running parallel loop iterations.

2.2 Runtime Systems for Speculative Parallelization

This thesis focuses on a specific kind of speculation, namely *memory speculation*. This kind of speculation optimistically ignores certain memory dependencies during parallelization (cf. Section 1.2). In the presence of irregular data structures, it is often a key technique for being able to extract any parallelism. This was discovered and described in many publications by different authors during the last decades (e.g. [6, 63, 110, 119]).

Memory speculation, however, requires *runtime support* to *dynamically check the validity* of the speculative execution. In order to detect overlaps in the accessed memory locations, every access to potentially shared memory needs to be tracked and compared against accesses by concurrent tasks. There are several options for implementing these runtime systems. The most important and most prominent options are discussed in this section.

2.2.1 Software Transactional Memory

Even before the potential of *thread-level speculation* for automatic parallelization was discovered in the late 1990s (e.g. [73, 110], see Section 2.2.4), a quite similar concept was introduced as *transactional memory (TM)* by Herlihy and Moss in 1993 [44]. It was designed as an alternative to lock-based programming, which is known to be prone to many kinds of errors, like *priority inversion*, where a high-priority task has to wait for a lower-priority task which holds a lock, *convoying*, where a task holding a lock is interrupted for a longer time by a page fault or any system call, and hence other tasks waiting for the lock cannot proceed either, and—probably most well known—*deadlocks*, which happen if two tasks take the same locks in different order.

Because of these problems, and also for efficiency reasons, researchers proposed to use *lock-free* (or *non-blocking*) concurrent data structures [35, 42, 46, 58, 111]. These are often hard to implement using the single word *compare-and-swap (CAS)* operation. TM provides the same semantics as CAS, but on a much larger number of independent memory locations. Therefore, one of its first uses was implementing

lock-free concurrent data structures. In this setting, critical sections are most often rather short, and just a few memory locations are accessed. Therefore, the original design only features these short critical sections.

Memory transactions are often compared to database transactions, featuring the *ACID* properties: *Atomicity* guarantees that each transaction appears to either have executed completely, or not at all. This is given for TM because each transaction either commits all its changes, or it rolls back, discarding all changes to visible state. In concurrent systems, atomicity also implies that no intermediate state is observable at any time. Here, the literature differentiates *weak atomicity*, where only concurrently executing speculative tasks are not allowed to observe intermediate states, and *strong atomicity* where this also extends to non-transactional code. The latter is hard to achieve without hardware support, since multiple independent memory locations cannot be written simultaneously in software. *Consistency* defines the property that each transaction transfers the system from one valid state to another valid state. In TM, this property strongly depends on the semantics of the individual tasks: If they are *consistent* under the *single global lock semantics*, however, TM guarantees consistency, too. *Isolation* means that there exists a total order in which the critical sections would have produced the same outcome if executed sequentially. This feature requires atomicity, but provides more guarantees. Especially, it requires the effects of all transactions to be applied to the global state in the end. The *durability* feature defines that state changes by committed transactions will be persistent even in the case of hardware failures or other events. It does not apply to TM since all changes remain in main memory.

Even though it was designed as an alternative to lock-based parallel programming, TM can also be used to implement thread level speculation (TLS) [66, 96, 97]. Since transactional memory in general does not impose any ordering between transactions executing in parallel, special care has to be taken by the generated code to ensure correctness. Some approaches require a TM system which provides a global *commit order*, others establish a commit order themselves. In the remainder of this section, we review the evolution and the state of the art in TM implementations in software. The next section will introduce hardware implementations and discuss their use for TLS.

After the introduction of the concept of *transactional memory*, Shavit and Touitou describe and implement the first pure software implementation in 1995, since then called *STM* [106]. It already features word-level conflict detection and hence can be used for any imperative language—in contrast to all the approaches for object-oriented languages, where transactional metadata can easily be stored in the object header [40, 41, 90, 121, 122]. The authors provide proofs for the correctness and the liveness of their implementation. However, it is specialized to so called *static transactions*, where the full data set the transaction operates on is known in advance, making it easy to privatize and access speculative state. As an example application, they implement a *k-word compare-and-swap* using their STM.

In 2003, Herlihy et al. describe the first *dynamic* STM implementation, called DSTM [43]. It supports an arbitrary and unknown number of objects to be accessed by each transaction. It is however still not tailored towards automatic application by a compiler, since objects accessed transactionally need to be accompanied by a *TMObject* object with the same live range, making this system an *object-based*

STM system. If transactions work on irregular data structures like trees or linked lists, these data structures need to be changed to also include the *TMObject* objects.

In 2006—after the rise of multi-core processors—several new designs and implementations were proposed: Saha et al. present McRT-STM [99], a dynamic word-based STM implementation which is executed inside the Multi-Core RunTime (McRT) environment. In contrast to DSTM, transactional metadata is not stored within individual objects, but in a global data structure indexed by the *cache-line address* of the accessed object. This allows for a straight-forward code instrumentation via a compiler, but raises other issues like *false sharing*, leading to *false conflicts*. McRT-STM allows for different STM configurations: In the *reader locking* configuration, each memory location is associated with a reader-writer lock, which is taken on each transactional access. In *read versioning* on the other hand, no locks are taken on reading accesses, instead the read version number is recorded and compared against the current state on commit. A second choice is *write buffering*, where speculative changes are stored in private memory and only written back during commit, versus *undo logging*, where changes are written to main memory directly and the original value is kept in private memory for restoring it during rollback. For different applications different configurations perform best, depending on the access patterns of the speculative tasks, but also on the execution platform. In their evaluation, read versioning and undo logging performed best. However, McRT-STM leverages the scheduler of the McRT system to increase performance and avoid deadlocks, so it can only be used within that system. McRT-STM also provides object-based conflict detection like the other approaches

mentioned before, but in a statically compiled language, which is not strictly object-oriented. This is achieved by modifying the McRT internal memory allocator, so it is only applicable to objects on the heap, and only works within the McRT execution environment.

Still in 2006, Riegel et al. [89] as well as Dice et al. [24] introduce the concept of a *global version-clock* (or *timestamp*) to efficiently re-validate the read set and to detect *read-after-write* (RAW) violations (missing an update from an already committed transaction). Just as Saha et al., they are using a *single global array of locks* for write locking and storing version numbers. For assigning locks to *memory stripes*, they generally use simple hash functions.

Dice et al. [24] implement these techniques in their *Transactional Locking II* (TL2) system, which since then serves as a reference implementation that many follow-up approaches compare against. In contrast to Saha et al. (see above), they are using write-buffering and commit-time locking. Write-buffering simplifies the rollback process, but introduces overhead at each memory load, since the write-log has to be searched for an entry before reading from main memory. Commit-time locking again reduces the rollback cost and ensures that write locks are held as shortly as possible, but delays the detection of memory conflicts and requires to re-validate the read set after acquiring all write locks.

Riegel et al. [89] call their approach *Lazy Snapshot Algorithm* (LSA) and provide an implementation called TinySTM [31, 32]. They use a global timestamp to establish a *validity range* for each transaction. This range is narrowed on a memory read which is younger (i.e. larger version number) than the start of the range. Once the validity range

becomes empty, the read-log is re-validated to make sure that the transaction operates on any valid "memory snapshot" (hence the name LSA). TinySTM can be configured for either undo-logging or write-buffering (also called write-through and write-back), and commit-time or encounter-time locking[1]. In the performance evaluation, TinySTM outperforms TL2 in all configurations, especially if the benchmark shows high contention rates. This is because encounter-time locking detects memory conflicts earlier and thus avoids useless work.

In 2007, Wang et al. [120] extend McRT-STM by a timestamp mechanism similar to that of TL2, and introduce language constructs for using STM in C and C++ programs. They also describe the compiler transformations and optimizations to generate efficient transactional programs.

In 2009, Mehrara et al. propose the STMlite system [66], which is the first STM system specifically targeted at *automatic parallelization*, in this case focusing on loops only. They use a central commit unit called *transaction commit manager* (TCM), which checks for conflicts between transactions. Additionally, each transaction stores *read-* and *write-signatures* similar to bloom filters [8], but—in contrast to earlier proposals [17]—implemented entirely in software. Those signatures are then also transferred to the TCM for conflict detection. This ensures fast transactional reads and writes, since only thread-local data structures are updated. In return, one single processing unit—the TCM—has to do all conflict checking, which might become a bottleneck. STMlite uses lazy updates (write-back) in combination with lazy conflict detection (at commit time). This adds the risk of

[1] Commit-time locking was added to TinySTM after the papers [31, 32] were published.

"zombie transactions", which are defeated by periodically checking the incomplete read- and write-signatures for conflicts with already committed transactions.

Even though STMlite specifically targets automatically parallelized programs, the specific requirements that those applications pose to STMs are not further investigated. In the evaluation, they use the standard STAMP [80] benchmark suite and several smaller applications from different domains, but it remains unclear how STMlite would perform on larger programs.

In the same year, Dragojević et al. present another word-based STM implementation called SwissTM [27]. It was designed with a focus on good performance on a broad range of atomic sections, especially long running ones. The authors claim that especially non-expert programmers and automatically parallelizing compilers might produce those large transactions. Similarly to TL2 and TinySTM, SwissTM uses a *global lock table* of fixed size to resolve read-after-write and write-after-write conflicts. Beside the usual STAMP benchmarks, the authors also evaluate SwissTM on Lee-TM [2], STMBench7 [37] and a red-black tree implementation. In these benchmarks, SwissTM performs better than any other tested STM system.

In 2010, Dalessandro et al. publish about an STM design without any *ownership records* (like for example locks), which are typically used in STM systems for tagging which transaction holds speculative writes on specific memory locations. The proposed *NOrec* [21] system instead validates the entire read set after each commit of any concurrent transaction, using *value-based validation*. They later extend their system [20] to also include *hardware transactions* utilizing the HTM

features of Sun's prototype Rock processor, and AMD's proposed *advanced synchronization features (ASF)*.

In the same year, Gottschlich et al. develop another implementation called InvalSTM [36]. Instead of validating the read set before committing, they use the opposite approach of invalidating concurrently executing transactions before committing any transaction. They use bloom filters to store the read and write sets, which enables efficient lookup and intersection. This approach is later extended by incorporating hardware transactions by Calciu et al. [14]. By some modifications to the STM system and careful design of the hardware transactions utilizing Intel's RTM technology, they allow for concurrent execution of hardware transactions and software transactions based on InvalSTM. Transactions are first executed in the HTM setting, and if this fails repeatedly, the STM is used as a fallback.

The STM^2 implementation by Kestor et al. [54] uses simultaneous multithreading to hide some of the STM overhead by offloading it to a sibling hardware thread. The main thread still manages the write set, because it needs to be traversed at each speculative read, but the validation of values read from main memory and acquiring ownership for written memory locations is performed by the sibling thread. Both are connected via a lock-free queue. Since both threads are pinned to neighbouring cores, they are likely to share most cache levels, such that communication is cheap. The evaluation shows that by removing part of the TM validation work from the application threads, STM^2 outperforms traditional STM systems for many applications.

Most STM systems do not discuss the problem of *commit ordering*, because it is not relevant for the targeted uses. Note that lock-based

synchronization does also not enforce any specific ordering. Whenever STM systems are used for speculative parallelization though, commit ordering becomes an issue ([23, 97]). It is often not solved by modifying the STM system itself, but by waiting for a certain program state before allowing a transaction to commit. This can by achieved by placing additional synchronization code just before each commit point.

2.2.2 Hardware Transactional Memory

Instead of implementing transactional memory in software, several designs for hardware-supported execution have been proposed. Already the initial description of the concept of transactional memory by Herlihy and Moss [44] suggests an implementation entirely in hardware. This is achieved by adding an additional first-level *transactional cache*, where each cache line is tagged with one of four *transactional tags*. Both caches (regular and transactional) are exclusive. Additional memory instructions are added for loading and storing memory transactionally. Each execution of one of these instructions automatically starts a transaction if none is executing yet. Separate instructions are provided for querying the current transaction status, aborting the transaction, or committing changes to main memory. Since all caches *snoop* on the memory bus, the state of the cache lines is updated whenever any processor core touches the same lines. Memory conflicts are hence detected *eagerly* at no additional cost; however, it is up to the software to check the transactional state frequently enough to detect conflicts and re-execute accordingly.

Rajwar and Goodman propose an approach called *Transactional*

Lock Removal [79], which executes traditional lock-based parallel programs in a hardware transactional memory system via a technique called *Speculative Lock Elision* (SLE) [78]. By eliminating a lock and instead treating it as the defining scope of a transaction, they transparently transform a program into a non-blocking speculative program. By serializing transactional tasks in the case of conflicts, they provide progress guarantees without any software back-off implementation. They propose to implement the buffering of speculative state completely in hardware, by utilizing the existing *store buffer* for delaying speculative stores until commit, and utilizing the *reorder buffer* together with register checkpointing for buffering speculative register changes. Hence they do not require any additional hardware, but extend existing structures by additional tags.

Although we do also wish for such a hardware mechanism, we do not see it coming yet. Lately, Intel added limited support for HTM in the Haswell architecture, called *transactional synchronization extensions (TSX)*. It supports SLE and is implemented very similarly to the design proposed by Rajwar and Goodman and later detailed by Steffan et al. [108, 109] and Steffan and Mowry [110]. Even though this looks promising, we quickly discovered that it is unusable for thread level speculation, as there is no ordering between the individual tasks. They also cannot be added in software, as this would require communication between the involved cores, and any communication by definition violates the constraints of a transactional task. Also, the amount of transactional memory is limited by the size of the first-level cache, and the execution of many instructions immediately leads to abortion of the task. This restricts the uses of TSX to short-running unordered transactions.

2.2.3 Thread Level Speculation

As mentioned in Section 2.2.1, *software transactional memory* (STM) systems can in general be used to implement *thread level speculation* (TLS). Since the characteristics of typical STM tasks greatly differ from those arising from speculative parallelization, and the latter impose more constraints on the ordering of transactions, specialized systems for TLS have been developed. The most promising ones make use of the *virtual memory* system implemented in the operating system with hardware support in most architectures. This section gives an overview of the origin and state of the art of these kinds of systems.

Similar to transactional memory, TLS can either be implemented entirely in software, or with hardware support. The first work describing speculative parallelization of loops with runtime checks was the LRPD test by Rauchwerger and Padua [88]. The main purpose of this work was removing dependencies by privatization and reduction recognition; any data dependence which could not be resolved via one of these techniques would cause sequential re-execution. All checks and rollback are executed in software. Three years later—in 2002—the first software-only dynamic TLS implementation was presented by Rundberg and Stenström, called S-TLS [92] and written in pure assembly. It uses shadow memory to track read and written memory regions and take ownership locks on them, and to hold the updated values for a fixed number of parallel tasks. This scheme was later improved by Cintra and Llanos [19] by significantly reducing the memory overhead, improving the access structures and eliminating the need for explicit locks. Years later, more advanced implementations

have been proposed for both write-back [116] and write-through [72] designs. The latter, called SpLIP, is particularly interesting because it updates memory in-place instead of buffering speculative stores, thereby avoiding most of the overhead of speculative loads. In the best case, this system performs much better than write-buffered systems. However, if rollbacks are expected to occur the performance drops dramatically, since rollbacks are much more expensive and also invalidate concurrent work which might have read the invalid memory update. Also, this system requires hardware which provides a sequentially consistent memory model, which is not provided by most of today's off-the-shelf processors.

Since all these software-only approaches introduce significant runtime overhead, other research made use of different hypothetical hardware extensions in order to speed up management of speculative data. Interestingly, implementing TLS in hardware was proposed already long before multi-core processors became mainstream, e.g. in the *Hydra CMP* [39] and others [107, 110]. The STAMPede project by Steffan et al. [108, 109] and Steffan and Mowry [110] proposes similar extensions to Rajwar and Goodman, but describe the extensions to the cache coherence protocol in much more detail. They extend the regular first-level cache with *speculative cache line states*, and add several new messages to the regular MESI cache coherence protocol for managing speculative accesses. These messages also carry an *epoch number*, which encodes the sequential order of the speculative tasks, making the system a real TLS system and not just HTM. As this approach does not target a specific architecture, they do not focus on the instruction set extensions, but only on the hardware and protocol extensions.

In 2006, Liu et al. published the POSH compiler [63], which aims to extract speculative parallelism to be executed in a hardware TLS system with simple *spawn* and *commit* instructions. One vehicle to implement these instructions in hardware is via *versioned memory*: Speculative tasks generate a new *version* of the memory, which can later be committed, or can easily be discarded. This can be seen as a restricted variant of *hardware transactional memory* which provides encapsulation and atomicity, but does not check for misspeculation. Hence these checks need to be done explicitly in software. If only control flow speculation is used (effectively ignoring the memory effects of certain code paths during analysis), a rollback is triggered whenever such a path is taken. This is implemented for example in Spec-DSWP [117], a speculative extension of *Decoupled Software Pipelining (DSWP)* [84].

Johnson et al. [51] provide performance measures of their parallelization approach on a simulated *speculative multi-core processor*. This hardware precisely detects true data dependencies at no cost and without limitations in the transaction size, so the reported speedup can merely serve as upper bound on the speedup to be expected on real hardware. Hertzberg and Olukotun [47] also evaluate on simulated hardware with full TLS support, but they describe the expected hardware extensions in detail and take care to make reasonable assumptions there. They utilize the first-level cache to store the speculative read set per processor core, and add an additional speculative store buffer for the write set. Whenever one of them overflows, a rollback is triggered. Also, speculative stores are broadcast within a *cluster* of four cores, sharing one second-level cache, in order to detect violations eagerly.

2.2.4 Virtual-Memory Based Memory Tracking

A third category of TLS systems is neither pure software, nor does it rely on special hardware. It utilizes the virtual memory system to separate speculative from committed state and track memory accesses for later validation. Since all modern processor architectures handle virtual address translation in hardware, this can be very efficient.

The idea of protecting individual memory pages and transferring their content and modifying ownership information in the page fault handler originates from earlier work on *distributed shared memory (DSM)*, starting in the mid 1980s. Li and Hudak [61] provide a nice overview of the early work in this area and the different design choices to implement it. In contrast to the hardware this thesis focusses on, their work assumes a cluster of workstations that communicate via network connections. Thus the transfer of a page between processing units takes much longer, leading to different design decisions. In order to keep track of the state of each page, they propose a *centralized* or *distributed manager*. They simulate their approach using different *memory page sizes* and argue that the best choice is system and application dependent. Fleisch and Popek [34] use the same idea but implement it directly in the operating system kernel. Keleher et al. [52] later implement the same protocol as a user-space library. Schoinas et al. [104] propose different schemes for improving the granularity of access control in DSM. One of the solutions instruments the program by inserting a lookup in a global data structure to determine the state of a memory block before each access to shared memory. In their evaluation, this software solution is up to two times slower than a simulated hardware-only solution.

More recently, Sadini et al. [98] implement a replicated-kernel operating system based on Linux to run on clusters of machines with different instruction set architectures (ISAs). The cluster should be transparent to the applications running on it, so they are provided with a shared memory view. This is implemented by a *page coherency protocol* similar to the MSI cache coherency protocol and along the lines of the already mentioned approaches. Sadini et al. implement this by extending the kernels `struct page`, which holds important information about the current state of each physical memory page. They add additional flags to store the owner and the replicate state of the respective page. Such changes can only be done if the whole kernel is recompiled, not by a kernel module.

Papadimitriou and Mowry were the first ones to use similar techniques in order to implement TLS, described in a technical report in 2001 [75]. Surprisingly, it did not get much attention in the community. The authors were involved in the hardware TLS implementation in the STAMPede project [108–110], which uses a single-chip multiprocessor with extended hardware and an extended instruction set to support TLS. Now they describe a TLS system which does not rely on any special hardware or compiler support, and can be integrated by just linking against a software library. The speculative memory region which is to be protected by the TLS system has to be allocated manually, however; thus the protection does not extend to arbitrary memory objects on the heap, stack or data segment. Individual *unix processes* are spawned to execute speculative tasks. By making the memory pages inaccessible in a speculatively forked process and installing a custom page fault handler, all pages read and written by a process can be recorded with only constant overhead per accessed

page. Conflicts are then checked at the granularity of memory pages (4 kB on most current architectures). Finished tasks are first validated, then put in a queue of completed tasks. Processes executing later tasks use this queue to validate their changes, and also to update their own memory view before executing the next task (hence memory changes are replayed by each single processor). The authors are aware of the problem of *false sharing*, leading to *false conflicts*, and propose to use *diffing* on changed memory pages to eliminate some *write-after-write* dependencies. As this does only work if no memory on the respective page was read by the task, this is no general solution for improving the granularity.

The first approach extending the idea of utilizing the virtual memory system to arbitrary memory pages is *behavior oriented programming (BOP)* by Ding et al. in 2007 [25]. Instead of replaying memory changes in all other processes, they copy back modified memory pages to the main process. BOP solves the problem of false sharing for global variables by allocating each of them on an individual page, but this comes with an increased number of page faults and added overhead for copying unused memory space. Even though BOP is not tailored to fork-join parallelization, it could potentially be modified for this use as well. A similar approach by a subset of the authors uses the same technique for *Fast Track* [53], where optimistically optimized code is executed in the main process, while the original code is executed in concurrent processes for validation.

Later, Berger et al. [4] describe a quite similar system used to detect and prevent concurrency errors in multi-threaded programs. By turning threads into processes, they achieve strong atomicity and avoid deadlocks, and by committing changes sequentially, they prevent

any data races. Raman et al. [81] also use separate address spaces to separate speculative states, but still track memory accesses explicitly and replay them in a central commit unit. Pyla et al. [77] use process-separation to support speculation in the form of different algorithms solving the same problem concurrently, and only committing the first one to complete. Kim et al. [55] describe a TLS based on memory page protection designed for clusters, with a dedicated validator and commit process.

2.2.5 The Importance of Granularity

Papadimitriou and Mowry [75] raised the issue of a coarse granularity implied by only being able to protect and observe memory accesses on whole memory pages. In their evaluation, they show how an increase in the block size translates to many more false conflicts being detected, and also more memory being copied during privatization of speculative state, and for committing successful transactions. They conclude that access tracking at finer granularities is needed, but that the overhead of such techniques would probably render them impractical for automatic parallelization.

Burcea et al. [12] evaluate several applications from the SpecINT benchmark suite. They track the number of tracking elements and the number of false conflicts for different granularities, and define the *ideal granularity* per code region as the coarsest granularity which does not cause any false conflicts. For the evaluation, they build on the STAMPede simulator [109] by Steffan et al. Similarly to Papadimitriou and Mowry, their numbers show that the share of false conflicts increases for coarser granularities while the number

of tracking elements decreases. Both effects strongly depend on the memory access pattern of the application. The ideal granularity per code region varies between 2 and 2048 bytes, and often varies heavily within one program. Therefore the authors propose to not only choose one granularity per program, but even adapt it for each speculatively parallelized region within a program.

Mannarswamy and Govindarajan [64] evaluate the effect of different granularities on the STAMP [67] benchmark suite. They modified the TL2 STM implementation [24] to support varying granularities within the same application. A static analysis then determines a suitable granularity per atomic section, based on the data structures involved in the computation. The compiler adds code to switch the global granularity setting before starting an atomic section. This switch has to be global since a different lock table is used for each granularity, so consistency is only guaranteed if all concurrently executing tasks use the same granularity. Hence a switch is only performed if no atomic sections are currently executing. This still ensures that in different phases of the execution different granularities can be chosen.

2.3 Conclusion and Open Issues

The need for speculative parallelization has often been demonstrated (cf. Section 2.1). The most successful approaches just assume a hardware TLS system tailored to the special needs of the respective approach, and get respectable speedup when simulating parallelized programs on such hardware. It is questionable, though, whether hardware vendors will ever provide the means assumed in these

approaches and whether the performance penalty will be as low as often assumed.

Several TLS approaches thus use the hardware features we have today. The simplest designs just record memory accesses explicitly by instrumenting the program (cf. Section 2.2.1). Such approaches, however, impose a substantial performance penalty, as we will investigate in Section 4.3. More advanced designs try to reduce overhead by using the virtual memory system to track and isolate memory changes (cf. Section 2.2.4). Often, however, the exact implementation of such approaches is not described in the papers, as they focus on parallelization or other techniques. This makes it hard to compare with them or use them for further research. Also, most of the systems are not publicly available.

Therefore, this thesis presents three easy-to-use open source solutions for virtual memory based TLS. One is implemented in user space only, the second one includes a Linux kernel module for maximum performance. The third one builds on the latter by augmenting it with instrumentation for more precise memory access tracking. This provides much better granularity (down to byte level), thus reducing the amount of false sharing and hence the amount of rollbacks being executed. These systems are the subject of this thesis and are described in the following chapters.

Chapter 3

The Sambamba Framework

In order to implement and evaluate one's own parallelization and speculation approaches, one typically extends an existing compiler. This allows to reuse its front end, back end and existing analyses and transformations. There currently exist two important compiler suites for C/C++ that are open-source and can be used for this: gcc and clang/LLVM. We decided for LLVM for several reasons: First, it has a much cleaner codebase, which makes it easier to understand existing code and build on it. Secondly, it consistently uses a single intermediate representation which is fully documented. Thirdly, it has an active community, which is willing to help should any LLVM-related problems arise.

The individual components we are developing should work together and be reusable. Speculative parallelization for example should be able to use any of the presented runtime systems during execution.

Therefore, we create our own framework to integrate all our components. We call it *Sambamba*, which is a Swahili adverb for *parallel* or *side by side*.

This chapter describes the general design of Sambamba, the structure of the output files and the internal phases during compilation and execution of the compiled programs. The design and development of the Sambamba framework is joint work together with my colleague Kevin Streit. Different aspects of Sambamba were already introduced in several publications [112–114].

3.1 General Design

Sambamba is split in two parts: a static part which consists of a number of analyses and code transformations, and a dynamic part which is used during program execution. The interface to the static part is the *sambamba* command line tool which is invoked like any other compiler (cf. Figure 3.1). The only difference is that it does not contain any front end. It expects linked and executable LLVM IR as input, meaning it should contain a *main* function, and all symbols must be resolvable either within the program itself or via referenced shared libraries.

In the default mode, *sambamba* outputs another LLVM IR file. The typical next step is to compile this file to object code and link it against the Sambamba shared library and others like pthreads and TBB. In order to facilitate this step, *sambamba* also contains a switch to directly produce a linked executable.

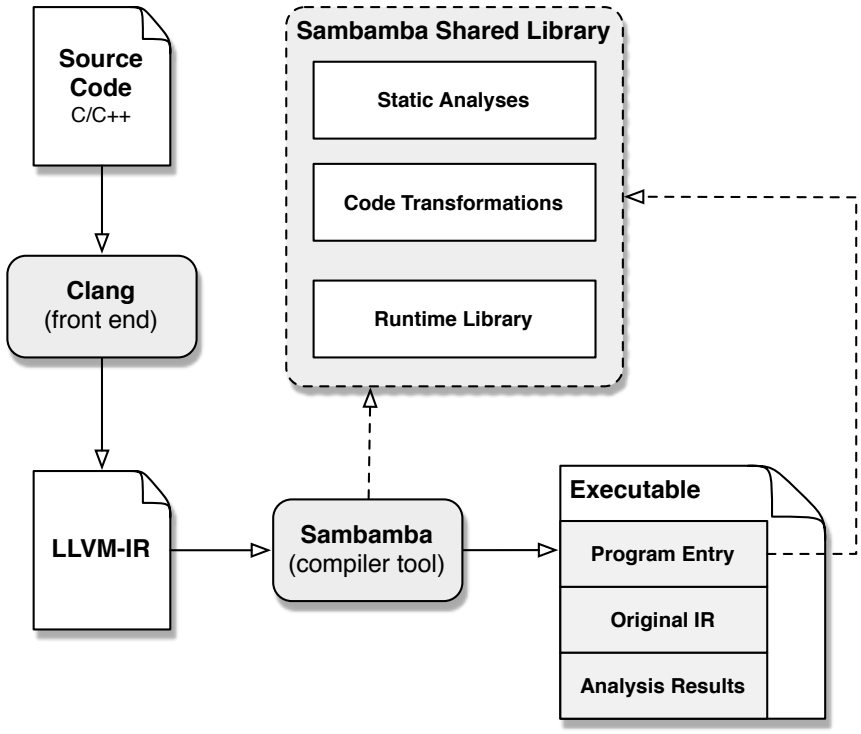


FIGURE 3.1: Overview of the compilation phase of Sambamba. The Sambamba compiler tool replaces your default back end: it takes LLVM IR as input and produces an executable file. This executable holds the original LLVM bitcode and analyses results. At Runtime, it calls back to the Sambamba shared library to trigger execution.

The original idea was for Sambamba to form an open platform which can be extended by an arbitrary number of *modules* providing the actual functionality like parallelization. Such a module would consist of a static part which is loaded and invoked during the compilation phase, and an optional dynamic part which is provided via LLVM bitcode. This dynamic part would then be copied into the executable file, and its entry point would be invoked after initializing the Sam-

bamba runtime system but before starting the execution of the actual program. It turned out that this flexibility was not needed later on and it complicates the reuse of functionality between modules. Therefore, we implement most functionality directly inside the Sambamba system itself (so it is part of the Sambamba shared library) and just invoke it during compile time or run time as needed. This also allows to change runtime functionality without recompiling all programs compiled with Sambamba.

The output of the *sambamba* compiler tool is not a static compilation of the original source code. Instead, the LLVM bitcode of the program is encoded as a constant in the *data* section, and the main function just contains a call to the Sambamba runtime system implemented in the shared library. It passes a pointer to the bitcode and some analysis results which are also stored in the *data* section. When executed, the runtime system will decode the passed data and initiate execution via a *just-in-time compiler* as described in Section 3.3.

3.2 Phases During Compilation

Figure 3.2 shows the phases of compilation with Sambamba. First, some preparation passes are executed, like constant folding, transforming certain memory operations to SSA operations, reducing computations by *global value numbering*, inlining or loop strength reduction. The exact set of optimizations can be modified with command-line switches. The purpose of these passes is to bring the input program to a form which reveals most parallelism, while still preserving most of the structure the programmer gave to the implementation. Experiments have shown that other transformations

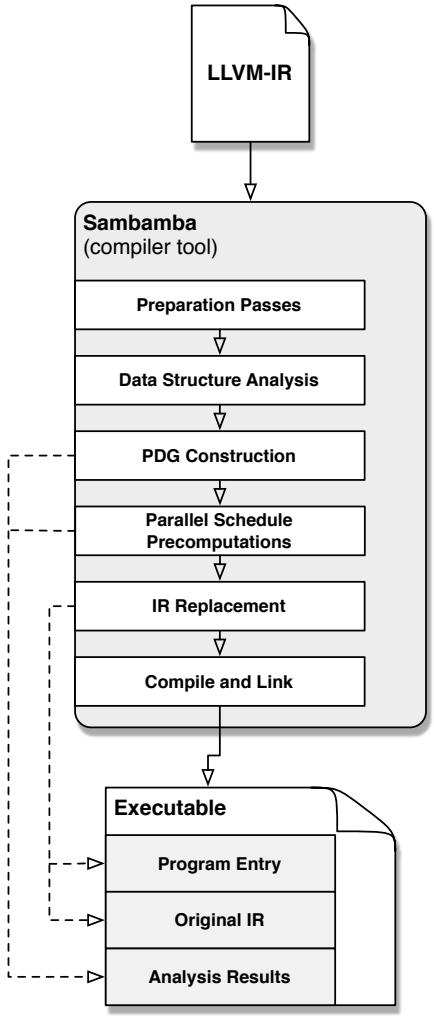


FIGURE 3.2: The steps executed during compilation with Sambamba.

like loop invariant code motion can reduce parallelism by introducing additional dependencies.

After these preparations, dependence information is computed by running the *data structure analysis* (DSA) [59]. This is a sophisticated

interprocedural points-to analysis which is both flow-sensitive and context-sensitive. It is able to prove far more memory references disjoint than state-of-the-art alias analyses shipped with LLVM. The result of DSA is the basis for computing a *program dependence graph* (PDG) per function. Since the PDGs are also needed at runtime, they are stored in the object file. To this end, Sambamba provides a so called *static data store* to store arbitrary statically computed values and make them available at runtime. At compile time, the static data store is just a map which is filled by various static passes with key-value pairs of type string.

Based on the PDGs, Sambamba now precomputes schedules for individual functions as described by Streit et al. [112]. Code or bitcode for these schedules is not generated yet. A schedule describes the order in which the basic blocks of the function should be executed, and contains fork and join points. Those schedules are also stored in the static data store and will be used at runtime to generate the actual parallel code.

At the end of this pipeline, the LLVM bitcode of the whole module is serialized in LLVM's binary bitcode format. Then, all code and all data are removed from the compilation unit. They are replaced by two constant strings containing the bitcode of the application and the serialized static data store, and a single *main* function. The code of this function just calls into the Sambamba runtime—implemented in the shared library—and passes the pointers to the bitcode and the data store. If requested by the user (see previous section), this new module is then translated to assembly using LLVM's assembler tool *llc* and finally linked against the Sambamba shared library to produce an executable binary.

3.3 Phases During Execution

When the compiled program is executed, its main method will call into the Sambamba runtime, passing pointers to the bitcode and the static data store holding analysis results. Sambamba's runtime system will then decode the bitcode. This original instance of the bitcode will always be kept, such that later re-compilation can start from the unmodified code. Also, the runtime interface to the static data store is initialized as follows. The compiler tool prepared the data such that it can be accessed efficiently without copying anything. The serialization of the static data store starts with a list of ⟨key_ptr, key_len, value_ptr, value_len⟩ tuples sorted lexicographically by the key string, followed by a concatenation of the actual key and value strings. This allows to look up elements using a simple binary search. Thus, the runtime interface to the static data store just stores a pointer to the list of tuples and the total number of elements.

After the input data is read, Sambamba proceeds by initializing the execution engine which will later execute the program using the integrated just-in-time compiler. Then, it initializes all the *runtime modules* which registered at a central registry in Sambamba. Most notably, this will trigger *parallel code generation* in the parallelization module. Based on the PDGs and the schedules read from the static data store, the parallelizer will generate bitcode for the concrete parallelization per function. Depending on whether speculation is involved and which speculation system was chosen from the command line, parts of this generated bitcode are then instrumented. For STM instrumentation, the code will contain *callbacks* into the runtime system for each single memory operation, and for setup and commit

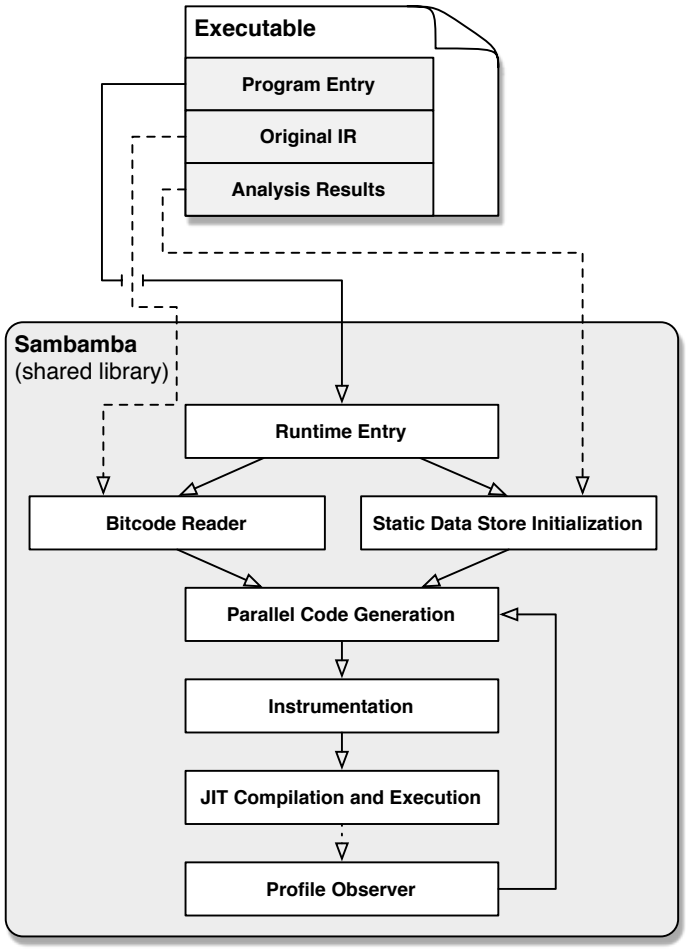


FIGURE 3.3: The phases executed at runtime of a program compiled with Sambamba.

of each transaction (see Section 4.2 for details). For the pure virtual-memory based approaches (cf. Chapters 5 and 6), no instrumentation is needed, but the code calls into the runtime to set up the task list, and to trigger its execution. For the virtual-memory based system with improved granularity, the code will again be heavily instrumented (cf. Section 6.2.3).

After all these transformations, the execution engine is instructed to run the main function. Whenever a function is entered for the first time, the just-in-time compiler will produce the respective machine code and then execute it.

The runtime system only becomes active when called from the compiled code, or via the profiler. The profiler performs light-weight sampling-based *execution time profiling* on a function basis. Profiling of individual functions can be enabled or disabled at runtime, typically triggered by the parallelizer. Initially and also whenever a function is replaced in the module which is being executed, the profiler checks whether profiling has been requested for this function. In this case, at function entry a new block is placed which checks whether the current run should be profiled, and if so, stores a monotonously increasing timestamp with nanosecond granularity. Also, all returns are redirected to a block which checks whether the first timestamp was taken, and if so gets another timestamp, computes the difference and updates the average execution time of this function using an exponentially floating average. The update is executed using a compare-and-swap operation, such that profiling is thread-safe. The flag whether the next run should be profiled is updated by a dedicated thread according to a defined sampling rate. This approach turned out to be the most efficient with respect to runtime overhead. For external functions however, this approach does not work since their implementation is not under the control of Sambamba. Therefore, if such a function should be profiled, all call sites are instrumented instead. This results in a larger code bloat and a scan of all functions in order to find all call sites. This is why we decided against using this approach universally for all profiling.

Chapter 4

Preparing STM for Speculative Parallelization

Software Transactional Memory (STM) is a technique to observe and isolate the memory operations of individual tasks, check for conflicts between concurrently executing transactions and either commit or roll back the changes. Hence it provides the basic functionality needed for automatic speculative parallelization. However, practical implementations of STM have not been implemented for this particular use case. This chapter describes the shortcomings of existing STM implementations (see Section 2.2.1 for an overview) and the modifications we did in order to use it in the context of automatic parallelization. Section 4.3 then evaluates the performance of an unmodified state-of-the-art STM implementation as well as our optimized implementation.

4.1 Changes in STM Design

We chose to base our work on the TinySTM system. In several performance evaluations [26, 31] it proved to be among the fastest systems. Also, it is freely available and provides an API which makes it easily usable for automatic transactification by a compiler.

This section describes the modifications we made to TinySTM in order to make it usable for TLS and to improve its performance when used in the context of automatic speculative parallelization.

4.1.1 Commit Ordering

In order to use TinySTM for speculative multithreading, we first had to add support for commit ordering (CO). This is essential in the context of speculative parallelization to guarantee conformance to sequential semantics. As of its newest version (1.0.5 at the time of writing), TinySTM has some support for CO, but for our use it is too limited: First, the sequence number of each transaction is only determined by its starting time, and this is nondeterministic if several threads are spawned at the same time. Second, the way it is implemented is problematic when executing long-running transactions: Before committing, TinySTM waits in a busy loop until all its predecessors have committed their changes. This design decision was probably made for very small transactions, were the overhead of taking a look or suspending the thread is larger than just waiting for the other task(s) to finish.

Therefore, we add our own CO protocol on top of TinySTM. This protocol assumes the dependencies to form a set of trees, i.e. each

task has at most one dependency. In *Sambamba*, we just build one chain of dependencies for all the transactions that might conflict with any other transaction in the same parallel section. The order of the tasks in this chain is determined by their original sequential order. The protocol is implemented by just adding a semaphore to each task, plus a pointer to the immediate predecessor task. Since *Sambamba* uses the Intel Threading Building Blocks (TBB) library [115] to implement parallel execution, we also use the semaphore implementation provided by TBB such that a waiting transaction can already start executing another waiting task. Before a transaction with commit dependencies is allowed to commit, it has to wait on the predecessor's semaphore. Upon completion, it signals its own semaphore to wake up the potentially waiting successors.

Note that our extension is not implemented within TinySTM itself, but in the surrounding compiler-generated code. As a consequence, it does not impact the performance of transactions which do not require commit ordering.

4.1.2 Hash Tables for Transactional Logs

Even though STM systems are designed to isolate potentially conflicting parallel tasks from each other, the workload they are exposed to is typically quite different from the one generated by an automatically parallelizing compiler. The benchmarks used to evaluate STM (mainly STAMP [80]) often spawn tasks which execute in parallel for a longer period, but the code section covered by the STM system constitutes only a very small portion of the overall running time

of the parallel tasks. Also, one parallel task not only executes one transaction, but often a large number of them.

In automatic parallelization on the other hand, a transaction covers an entire parallel task, and the goal of the parallelizer is typically to produce long-running tasks in order to reduce synchronization cost and other task management overhead. Therefore, the design decisions taken by state-of-the-art STM systems have to be revised in this setting.

Looking into the implementation of modern STM systems (McRT-STM, TL2, TinySTM), we observe that all of them use the most simple data structure for storing the read and write logs: an array of structs. McRT-STM organizes this data structure as a *sequential store buffer* (SSB) according to Hosking et al. [48]. While this allows for a more efficient overflow detection, the data is still stored in an unordered array and the runtime for a lookup is not improved.

Even the STMlite system, which is specifically designed for executing automatically parallelized code, uses arrays for the read and write logs. In order to reduce the lookup cost, this systems adds a hash map based cache to the write log. This cache contains the latest written addresses and values, and helps reducing the overhead for some cases. Additionally, they use *signatures* similar to bloom filters [8] in order to prove transactional memory effects disjoint. Again, this only works reasonably well if the sets remain small.

When the read or write sets become very large, these data structures with linear lookup cost quickly become a bottleneck. At the scale of STAMP programs, where each task only updates a very small number of memory locations, an array might indeed be the fastest

possible data structure. For sizes growing to hundreds of elements or far beyond, this choice is questionable though. Remember that a write-buffering implementation needs to traverse the write set for each transactional read, in order to check for an updated value by the transaction itself. Also, the avoidance of duplicates in the read and write set needs linear time per update, and is essential to avoid running out of memory when executing large transactions. Together, this leads to an overhead of transactional read and write operations which grows quadratically in the number of elements in the respective set. We will evaluate the overhead caused by these data structures in Section 4.3.

In order to improve the lookup cost for large sets, the use of a *hash table* seems natural, and has already been proposed by Harris et al. as an optimization for an object-based STM implementation [41]. We thus replaced the write set by a hash map implementation. Since a lookup in the write set is performed on each transactional read *and* write operation, this should reduce the overhead significantly. In TinySTM, each write log entry already contains six fields, and is padded to 64 byte. The padded space is sufficient to additionally store a pointer to the next entry in the same hash table bucket, thus we implement a *chained hash table*. The hash function (mapping each memory address to a hash table entry) has to be efficiently computable, but also distribute addresses evenly across the table. In line with these requirements, we chose the following hash function with N being the current size of the hash map:

$$hash(a) := [\ (a \gg 3) \oplus (a \gg 8)\ ] \bmod N$$

In our implementation, N is chosen to always be a power of two,

allowing to efficiently compute the modulus by using a *bitwise and* operation. Since the STM system only permits aligned accesses, in a 64-bit environment the three least significant bits are always zero, hence they are ignored by the hash function. The goal of also incorporating some more significant bits is to ensure that also memory accesses with a constant stride are mapped to the full range of the hash table.

In the next step, we also implement the read set as a hash table. Since a read set entry only contains 16 bytes, adding a `next` pointer would double its size (including the padding). Thus we decided to implement an *open-addressed hash table* with linear probing instead. This way, the definition of a read set entry stays the same and the lookup code requires only a marginal change: Instead of searching from the beginning until the searched entry or an empty slot is found, we now start at the offset which the hash function computes. This linear probing also provides a good data locality during lookup operations.

4.1.3 Adaptive Initial Sizes of Hash Table

Both of the hash tables as described in the previous section are initialized to a size of 16, and once filled to 75 %, the table is resized by a factor of two, and all elements are copied to their respective bucket in the new table. Even though the cost for this operation amortizes over the number of elements added, resulting in just a logarithmic overhead per element added, experiments show that some programs spend more than 90 % of their execution time in the resize operation. Especially if the data structure does not fit into

the processor's caches any more, the resize seems to put too much pressure on the memory bus. We aim to reduce this cost by choosing a better initial size of the hash table. To this end, each application maintains two exponentially smoothed averages of the final size of its read and write sets, respectively. These averages are updated on each transaction commit according to the following formula:

$$update\_avg(old, new) = \begin{cases} old * \frac{15}{16} + new * \frac{1}{16} & \text{if } new \leq old \\ old * \frac{3}{4} + new * \frac{1}{4} & \text{if } new > old \end{cases}$$

$$avg\_read = update\_avg(avg\_read, current\_read\_size)$$

$$avg\_write = update\_avg(avg\_write, current\_write\_size)$$

This definition of the *update_avg* function gives more weight to values that are larger than the current average than to smaller values. This results in the average being shifted towards the maximum observed value during the last few transactions. This is especially useful if multiple transactions of different size are executed in an interleaved manner. In this case, all the transactions would start with a hash table big enough to hold the largest occurring sets, or only requiring one resize operation. We pay for this with a larger overhead for the smaller sets, but this overhead is much less than having to resize large hash tables repeatedly. One alternative would be to store the average read / write set size not globally per application, but per parallel section, or even per task within each section. In experiments however, the performance improvement was negligible, so we stick to one average per application. Since many of our benchmark programs only execute a small number of transactions, we implemented persistence of these values in order to start with meaningful values also for the

first transactions executed. The average values are stored in a simple database in the user's home directory, using the *module id* generated by LLVM as the identifier for the application. This ensures that the average not only persists across individual executions of the same binary, but also across recompilation.

On transaction initialization, the latest computed average is increased by a factor of $\frac{1}{75\,\%} = \frac{4}{3}$ in order to accommodate for the maximum fill rate of 75 %, and then rounded up to the next power of two. This size is then used as the initial size for the read and write set, respectively:

$$initial\_read\_set\_size = 2^{\lceil log_2(avg\_read\_set\_size * \frac{4}{3}) \rceil}$$
$$initial\_write\_set\_size = 2^{\lceil log_2(avg\_write\_set\_size * \frac{4}{3}) \rceil}$$

4.1.4 Hopscotch Hashing

Even when using the adaptive initial size as described in the previous section, we still observe program runs that spend most of their time in transactional read and write operations. Profiling and investigating this artifact revealed that this stems from very unbalanced hash tables: We observe that for some programs, the read memory addresses are clustered to a small number of buckets in the hash table. This basically leads to a linear lookup and insertion time, since the linked list of elements in the respective bucket has to be traversed each time.

This obviously is a problem of the used hash function, so we tried different shifting widths and measured the performance influence. None of the options we chose provided a consistently better performance for all programs and the whole range of reasonable input sizes. We spot the problem in the memory access patterns induced by some of

the benchmarks (see Section 5.3.2 for more information about the characteristics of these benchmarks): The *LU decomposition* and the *blocked matrix multiplication* for example work on differently sized blocks of a large matrix. These blocks form a *stride* of the consecutive memory of the matrix, and depending on the size of the blocks, the presented hash function will always map them to the same hash table bucket if the table size N is not increased proactively.

One option to overcome this problem would be to use a significantly better hash function providing a *uniform distribution* independent of any pattern in the input values. One could even use a *cryptographic hash function*. This would introduce another significant overhead though. So we decided to replace the whole algorithm to trigger a hash map resize operation. But since detecting that buckets exceed a certain fill rate would require additional resources to be spent on each insertion into the map, we decided to replace the whole hash map implementation instead. In 2008, Herlihy et al. introduced an interesting new approach called *hopscotch hashing* [45]. This hash map by design provides good data locality for both lookups and insertions. Each element is inserted within a certain neighborhood around the bucket selected by the hash function, and the table is increased when this neighborhood is fully filled and no entry can be moved out. This allows for some hash collisions to occur, but if too many values are mapped to the same or neighboring buckets, a resize operation is triggered.

We changed the hash map implementation for read sets to use this approach, and managed to keep the allocated size of read set entries unchanged, even though it now contains additional bookkeeping information for the hopscotch hash map. We achieved this by replacing

a pointer into the *locks* array by an index into the array, and using the remaining bits to store the *hop info*. We found the neighborhood size $H = 8$ to provide a good balance between locality and fill rate of the map. As the write log was performing well, we kept the chained hash table implementation for this data structure.

4.2 Instrumentation

TinySTM provides a C interface with the functions as listed in Table 4.1. Calls to these functions are directly instrumented into the code. The implementation itself is not compiled and linked into the shared library of Sambamba (see Chapter 3). Instead, the LLVM bitcode of the different parts of TinySTM is linked together and compiled as a string constant into the Sambamba shared library. The runtime system (see Section 3.3) then decodes this bitcode and links it into the application before the actual transactification. This allows to apply a number of interprocedural optimizations to the transactified code, especially to inline the STM functions and then optimize for the specific call sites. Especially the *stm_load_u8*...*stm_store_u64* functions contain different code paths for different alignments, falling back to the *stm_load_bytes* function if the address is not aligned properly. This check can often be eliminated by utilizing the alignment information from the replaced *load* or *store* instruction.

Now each transaction is instrumented by placing code at the beginning and the end of the respective region, and also instrumenting all instructions in-between. At the start of the transaction, code is added to call the *stm_start* function and then *stm_get_env* to obtain a pointer to the buffer where the *longjmp* data should be stored for

TABLE 4.1: C interface of TinySTM, used for automated instrumentation of speculatively parallelized regions.

stm_init	initialize the STM infrastructure; resets the locks array, installs signal handler, reads average read and write set size from previous runs
stm_exit	cleanup STM data before exiting; stores current average read and write set size to disk
stm_get_env	get a buffer for storing the *longjmp* environment
stm_start	start a transaction by initializing thread-local bookkeeping data structures
stm_commit	commit a transaction; check for memory conflicts and write back data
stm_load_bytes	transactionally load an arbitrary and/or dynamic number of bytes into a thread-private buffer
stm_store_bytes	transactionally store an arbitrary and/or dynamic number of bytes from a thread-private buffer
stm_load_u8	transactionally load one byte (8 bit) of memory
stm_load_u16	transactionally load two bytes (16 bit) of memory
stm_load_u32	transactionally load four bytes (32 bit) of memory
stm_load_u64	transactionally load eight bytes (64 bit) of memory
stm_store_u8	transactionally store one byte (8 bit) of memory
stm_store_u16	transactionally store two bytes (16 bit) of memory
stm_store_u32	transactionally store four bytes (32 bit) of memory
stm_store_u64	transactionally store eight bytes (64 bit) of memory
stm_store2	transactionally store up to one machine word (32 or 64 bit) of memory by specifying an aligned address, the new word and a mask encoding which bits to update
stm_set_bytes	transactionally fill a dynamically-sized memory range with a constant value
stm_memmove	transactionally copy a dynamically-sized memory region; source and destination may overlap
stm_malloc	transactionally allocate memory on the heap; is automatically freed on rollback
stm_calloc	transactionally allocate zeroed memory on the heap; is automatically freed on rollback
stm_free	transactionally free memory; will be deferred until commit
stm_abort_externalcall	abort the current transaction because an external function would be called in the original code

executing a rollback. Then a call to *setjmp* is added to fill this buffer. The *stm_start* function returns a pointer to the *stm_tx* struct holding all the bookkeeping data for the current transaction. This pointer is passed to all functions called in the transactional code. At the end of the transaction *stm_commit* will be called, passing the same pointer.

Some instructions in the code within the transaction are instrumented based on the type of the instruction as described in the following:

load A memory load instruction which is not statically known to access the top stack frame is replaced by one of the *stm_load_** functions, depending on the size of the loaded value. If this size is precisely 8, 16, 32 or 64 bit, then the respective function is used. Depending on the alignment and the size of a machine word, those functions may redirect to the *stm_load_bytes* function. If none of those functions matches the loaded size, then respective stack memory is allocated and the *stm_load_bytes* function is called directly to load from main memory to the stack slot. Then, a load instruction of the original type loads from the stack slot.

store Similar to a *load* instruction, we first check whether we can use any of the optimized functions to store a value of size 8, 16, 32 or 64 bits. These functions again redirect to the *stm_store_bytes* function if the alignment seen at runtime is not sufficient. Otherwise if the loaded memory is less than a machine word, the *stm_store2* function is called. If statically none of the fixed-width store functions matches, we call *stm_store_bytes* directly.

function call For function calls, we distinguish between three cases: Direct function calls, indirect function calls, and calls to intrinsic functions of LLVM. For all three though, we first check whether instrumentation is actually needed. If the call site or the called function or intrinsic is labelled with the *readnone* attribute, we totally ignore the function call. Also, we whitelist some of the intrinsic functions like *lifetime_start*, *lifetime_end* or *objectsize*, since they will not be translated to any instructions at runtime, and hence have no memory effect. We also whitelist functions from the C standard library like the trigonometric functions and other mathematical functions without (relevant) memory side effects. Apart from this, we instrument function calls as follows:

direct calls Direct calls are calls where the callee is statically known. If the called function is defined within the module itself, we also know the bitcode of this function. In this case, we look up whether we already transactified the called function, and if not, we (recursively) create a transactified version of the callee. The call site is then redirected to this transactified version. A pointer to the *stm_tx* struct holding all transactional metadata is passed as an additional argument to this function.

There are some externally defined functions for which special STM wrappers exist: *malloc*, *free*, *calloc* and *realloc*. For these, we just redirect to the wrapper.

In all other cases—if no definition is known for the function and no special wrapper exists—we replace the call by a call to the *stm_abort_externalcall* function, which aborts

the current transaction and passes a special value encoding the reason for the abort.

LLVM intrinsic calls There is a small number of memory-related intrinsic calls, for which special instrumentation is implemented: *memcpy*, *memmove* and *memset*. The first two are redirected to the *stm_memmove* function, the latter to *stm_memset*. All other intrinsics which are not known to be side effect free will trigger an abortion.

indirect calls For indirect function calls, the callee cannot be determined statically. In most cases, a pointer to the function is loaded from memory, either explicitly, or because of the implementation of dynamic dispatch in C++. We resolve indirect calls by managing a per-call-site cache mapping function pointers to their transactified version. To this end, we introduce a new static variable per indirect call site, pointing to an array of function pointers. Initially, all those cache-pointers point to an array containing only null pointers. If the respective call site is executed, the inserted code scans through the array to find an entry at an even position that matches the function pointer that would originally be executed. If it finds such an entry, it uses the next (odd) entry as the pointer to the transactified function. If it encounters a null pointer at an even position before finding a matching entry, it calls into a runtime function which resolves the function to be called based on the function pointer, and creates a transactional version of it if possible. The runtime function then updates the cache and potentially also the cache pointer if it still pointed to the shared null-array. If

no transactional version of the function can be created—if the function is external or the pointer is invalid—, then the *stm_abort_externalcall* function will be called directly by the runtime function. This keeps the amount of code to be inserted per call site small. When the runtime function returns, it also returns the resolved function pointer, such that no additional scan needs to be performed for this call. In any case, the inserted code just calls the transactified version—either found in the cache or returned by the runtime function.

4.3 Evaluation

In this section, we evaluate the performance of the original TinySTM system, as well as the version with our modifications, which we call TinySTM$^+$. For both systems, we measure the overall runtime of automatically parallelized and instrumented programs. Additionally, we provide a breakdown of the different sources of overhead.

4.3.1 State-of-the-Art Performance

For evaluating the performance of STM on automatically parallelized programs, we chose the programs from the *Cilk* [9] example suite. For those programs, we know that there exists a substantial amount of parallelism, and it is easy to reduce the programs to their so called *serial elision*, which is a fully sequential version obtained by serializing all *Cilk* program parts. We then ran *Sambamba* on these

sequential programs to search for automatically parallelizable code regions.

The *Cilk* programs contain recursive algorithms that pass pointers or values calculated on passed pointer values to subroutines. This often makes it impossible for the parallelizer to prove memory accesses disjoint statically. Hence, speculation is needed in order to automatically parallelize those programs. We identified eight programs from the *Cilk* suite containing at least one speculatively parallelizable region. For each such region, we automatically generate parallel code and instrument the speculative tasks as described in the previous section. In case of a memory conflict between two speculative tasks, the later task (according to the original sequential order) and all subsequent tasks roll back and re-execute sequentially.

In order to assess the performance of a state-of-the-art STM system on the large transactions that result from automatic parallelization, we configure TinySTM for *write-back* with *commit-time locking* (cf. Section 2.2.1). Commit-time locking is the only choice in our setting because when waiting for a commit-predecessor to finish (see Section 4.1.1), the thread should not hold any STM lock yet. Otherwise a high risk of deadlocks would arise. Write-back has been reported previously to perform better than write-through [32], especially when rollbacks are expected to happen.

Running the speculatively parallelized programs from *Cilk* using this TinySTM configuration, we were surprised about the performance impact: *None of the eight programs, which ran between 1.2 and 49 seconds sequentially, finished within a timeout of 12 hours.* For all but the longest-running *matmul* program this means a slowdown

of at least 2000×. We re-ran *matmul* with a timeout of 30 hours to confirm that also in this case the execution time of the parallel program instrumented with the original TinySTM implementation is at least 2000× the sequential runtime.

We manually checked that the code transformation and the instrumentation was correct, and that no deadlock occurred at runtime. In the following, we present an in-depth analysis of the overhead we measured in our benchmarks.

4.3.1.1 Overhead Breakdown

The usage of a software transactional memory system obviously introduces overhead to the execution of the application. This overhead can be dissected into the following categories:[1]

1. **Transaction Startup.** Starting a new transaction (i.e. entering an atomic block) requires setting up bookkeeping data. This includes initializing empty data structures like the read log, write log or undo log, saving program state for rollback, and fetching the value of the global clock.

2. **Memory Read.** If a write-buffering (i.e. write-back) approach is used, then on each memory read the write log has to be searched for an earlier write to the accessed location. Also, a new entry in the read log is allocated and filled, which implies updating several memory locations. A time-based implementation also fetches and stores the timestamp of the last update

[1] Depending on the type of the STM system, some of the listed work might be executed in different stages; the listing gives a general overview though.

of this location (i.e. its version) in order to detect inconsistent memory reads.

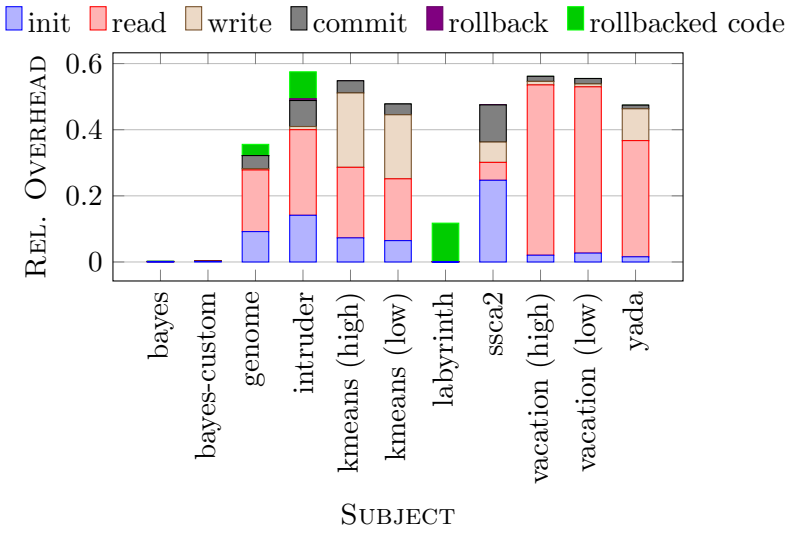
3. **Memory Write.** In case of encounter-time locking, the lock associated with the accessed memory address is acquired. Additionally, an entry in the write log is allocated and filled.

4. **Commit.** During commit, typically the read set needs to be revalidated, and memory data might need to be written back. Also, locks are taken and/or released using costly atomic instructions like compare-and-swap. Finally, memory used for bookkeeping needs to be reset or released.

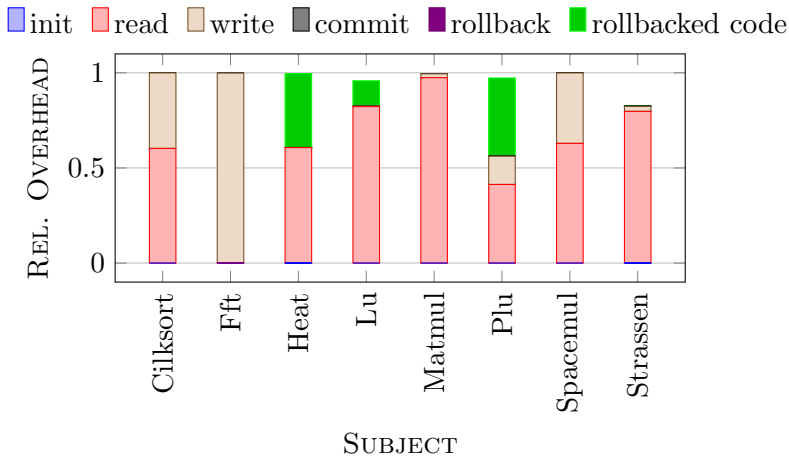
5. **Rollback.** The cost of a rollback is largest if the undo-logging approach is used, because the original values need to be restored in memory. Also, if encounter-time locking is chosen (which is mandatory for undo-logging), then also all acquired locks are released in case of a rollback. Additionally, the bookkeeping data of the transaction is reset, just like on transaction startup.

6. **Superfluous work.** A rollback not only implicates the overhead for executing the rollback as described above. It also renders all the work done since the transaction started useless—except possibly filling some cache entries, thus decreasing the number of cache misses during re-execution. Often however, the time for re-executing the transaction by far outweighs the immediate rollback cost.

The relative influence of each of these factors strongly depends on the target application and the memory access patterns within atomic regions. We measured them on a system with 8 GB of RAM and



(A) Overhead on the STAMP suite.



(B) Overhead on automatically parallelized *Cilk* programs.

FIGURE 4.1: Breakdown of TinySTM overhead on different programs from STAMP and the *Cilk* example programs, executed using four threads. Numbers are relative to total execution time. On STAMP, the overhead is between 0.003 and 0.58, and distributed among different overhead sources. On the *Cilk* programs, nearly all execution time is STM overhead, mostly in transactional *read* and *write* operations.

an Intel i7-870 processor with four cores running at 2.93 GHz and supporting hyper-threading. Turbo Boost mode was disabled during the execution of all experiments. We ran a Linux kernel 4.1.12 with glibc 2.21.

In order to influence the program's performance as little as possible, the measuring method has to be very lightweight. We decided against a sampling-based (statistical) approach, since we need to differentiate time in rollbacked code from time in productive executions, and this is only decided on commit or rollback. Therefore, we instrumented the code to measure the wall clock time spent in the TinySTM code parts of interest. We use the POSIX `clock_gettime` function with clock id `CLOCK_MONOTONIC` to get a monotonically increasing time stamp with a granularity of one nanosecond (ns). On our system we measured 6 nanoseconds for this function to execute. As the time span to measure is between a tenth of a microsecond and several microseconds, this overhead is tolerable. Getting the per-thread cpu time would have taken more than 150 ns, thus would have introduced too much bias to the measurement. The accumulated times are stored in the thread-private transactional metadata in order to avoid costly atomic memory instructions. Only after commit or rollback, the times are added to a global data structure.

All tests have been executed with four parallel threads, so that no transactions have to share a core for execution. Each test was executed ten times on an idle machine, and the median of the recorded values is reported.

For STAMP—a traditional benchmark for STM performance, see Section 2.2.1—the breakdown is shown in Figure 4.1a. For each of the

six overhead categories, the time is measured and shown in relation to the overall execution time of the program. First of all, we observe that over all programs, the relative amount of time spent in transactional operations is below 60 %. This translates to a theoretical slowdown factor of 2.5× or below, which conforms to the performance numbers reported in earlier papers [26, 67, 89]. While some programs show a mostly even distribution of execution time in the different STM functions, most of them are read-dominated, especially the *vacation* program.

Figure 4.1b shows the overhead breakdown for the speculatively parallelized programs of the *Cilk* suite. Remember that these programs did not finish within a timeout of twelve hours per execution. We report the statistics collected up to the point of abortion due to the exceeded timeout. The distribution of the overhead for these programs looks quite different than for STAMP: First, we see that each program spends nearly all of its execution time in STM functions, thus little productive work is executed. Also, only three of the programs show significant overhead due to rollbacks, so the huge overhead is not a problem of misspeculation. Apart from rollbacked code, the only noticable bars are transactional *read* and *write* operations, which are the real source of the observed slowdowns. As the rollbacked code was also executed transactionally, the distribution within that part is probably similar. Thus nearly all the execution time is spent for transactionally loading and storing memory.

The distribution of overheads differs a lot between *Cilk* programs and STAMP programs. The cause for these shifted overhead characteristics has to be related to the entirely different properties of the executed transactions, like for instance different memory access pat-

TABLE 4.2: Characteristics of the STAMP benchmark suite programs usually used to evaluate STM performance. Note the small fraction of time spent inside each transaction, and the low number of read and write accesses within transactions.

Benchmark	cpu time [s]	avg tx time [µs]	transactions	rollbacks	avg read set	avg write set
bayes	0.5	19.1	528	7	7.0	1.0
bayes-custom	0.3	17.5	516	15	7.0	1.0
genome	14.6	4.6	2.5e6	3.5e3	30.0	0.0
intruder	93.8	2.7	24.7e6	1.2e6	21.0	1.0
kmeans (high)	19.6	3.8	4.1e6	0	24.0	12.0
kmeans (low)	54.3	3.9	9.9e6	0	24.0	12.0
labyrinth	73.9	60.0e3	1.1e3	48	171.0	168.0
ssca2	42.7	0.7	22.4e6	99	0.0	1.0
vacation (high)	112.0	25.8	4.2e6	0	387.0	6.0
vacation (low)	81.4	18.5	4.2e6	0	280.0	4.0
yada	228.1	0.09	2.4e6	0	19.0	10.0

terns. In order to check this hypothesis, we also collected quantitative statistics of the executed applications. The result of this analysis is shown in Tables 4.2 and 4.3. In contrast to Figure 4.1, these tables show absolute numbers, like the average time spent in one transaction, from initialization to commit or rollback (labelled "avg tx time"). This time includes all transactional read and write operations. We collected all numbers using TinySTM$^+$, our improved variant of TinySTM. Otherwise, the programs would not have terminated and

TABLE 4.3: Characteristics of the *Cilk* example programs, automatically parallelized and instrumented using STM to guard against misspeculation. In contrast to STAMP, most of the time is spent in long-running transaction, with lots of memory reads and writes. Comparison of quantitative and runtime characteristics of the STAMP and *Cilk* programs we used to evaluate STM performance. STAMP spends most time outside of transactions, and transactions are short and with a very low memory footprint. In *Cilk*, only very few transactions are executed, but those are long-running and the touched memory regions are orders of magnitude larger.

Benchmark	cpu time [s]	avg tx time [s]	transactions	rollbacks	avg read set	avg write set
Cilksort	145	4.5	13	5	645.3e3	6.2e6
Fft	67	2.1	18	13	89.9e3	405.9e3
Heat	32875	108.4	202	101	346.5e3	1.4e6
Lu	11694	15.4	756	3	6.8e3	10.9e3
Matmul	1558	329.7	3	1	502.1e3	666.7e3
Plu	8683	0.4	13707	4023	3.4e3	10.4e3
Spacemul	172	5.9	20	6	78.6e3	3.5e6
Strassen	3407	0.0	951242	10050	634.0	1.0e3

we would not have gotten complete results.

For STAMP (Table 4.2), this time is mostly in the order of microseconds, where the applications typically run for several seconds. Thus, transactions finish shortly after they have started. In return, most STAMP programs execute millions of transactions. The exceptions are those three programs which also spend very little execution time

in transactional contexts (compare Figure 4.1), namely the two *bayes* programs and *labyrinth*. *labyrinth* on the other hand executes by far the longest transactions, they run for about one tenth of a percent of the execution time. However, within the transactions, most work is executed directly in main memory without making use of the STM system. Thus, the overhead per read and write, and also the size of the read and write set, remain very low. In summary we see that the STAMP programs execute only very small transactions with a minimal memory footprint—often just a handful of memory locations are touched.

The same measurements of characteristics for the *Cilk* programs are presented in Table 4.3. First, note that the unit of the average transaction time has been changed from nanoseconds to seconds. This already shows the huge difference between those test suites. A similar difference can be observed in the memory footprint of transactions. The *Cilk* programs touch thousands up to millions of memory locations. The program with the smallest footprint is *Strassen*, which implements the Strassen algorithm of matrix multiplication. Here, the read set grows to 634 entries on average, which is nearly twice the size of the largest read set average seen in STAMP, and the write set grows to 1013 entries—still six times the largest write set seen in the STAMP executions.

The largest write sets with 6.2 million entries on average are created by the *Cilksort* program. These are more than 36,000 times larger than the largest average write sets of STAMP. These large data sets of course post completely different requirements for the data structures.

4.3.2 Improved Implementation

This section evaluates how the changes described in Section 4.1 affect the performance of TinySTM. We investigate this both on automatically parallelized *Cilk* programs, as well as the STAMP programs, in order to check that there are no severe regressions for those small transactions where TinySTM already performed well.

Figure 4.2 compares the runtime of different variants of TinySTM. The blue bar shows the original implementation. Bars reaching below the vertical limit of the plot represent executions running into the timeout of at least twelve hours. As explained in Section 4.3.1, this means a slowdown of more than 2000× for each program. The red bar shows the performance if hash sets are used for both the read set and the write set. In this configuration, most of the programs run to completion within the timeout. The overhead over sequential execution is still tremendous, though. The third column adds the feature of storing the final sizes of the read and write sets, and using those numbers as initial sizes for new transactions. This change generally improves the performance, but its effectiveness differs between applications: While *Fft* and *Lu* improve by about 40 %, other applications like *Cilksort* or *Spacemul* improve by a factor of 100× or more. This effect is only partly caused by the reduced amount of re-hashing, involving expensive copying of the whole set. Initializing the hash set to a much larger size also reduces the risk of hash collisions. Another way to mitigate hash collisions or clustering of hash table entries is hopscotch hashing, as introduced in Section 4.1.4. By enlarging the hash table based on *local fill rates* within a certain neighbourhood, it is another solution to hash collisions that only occur for small hash

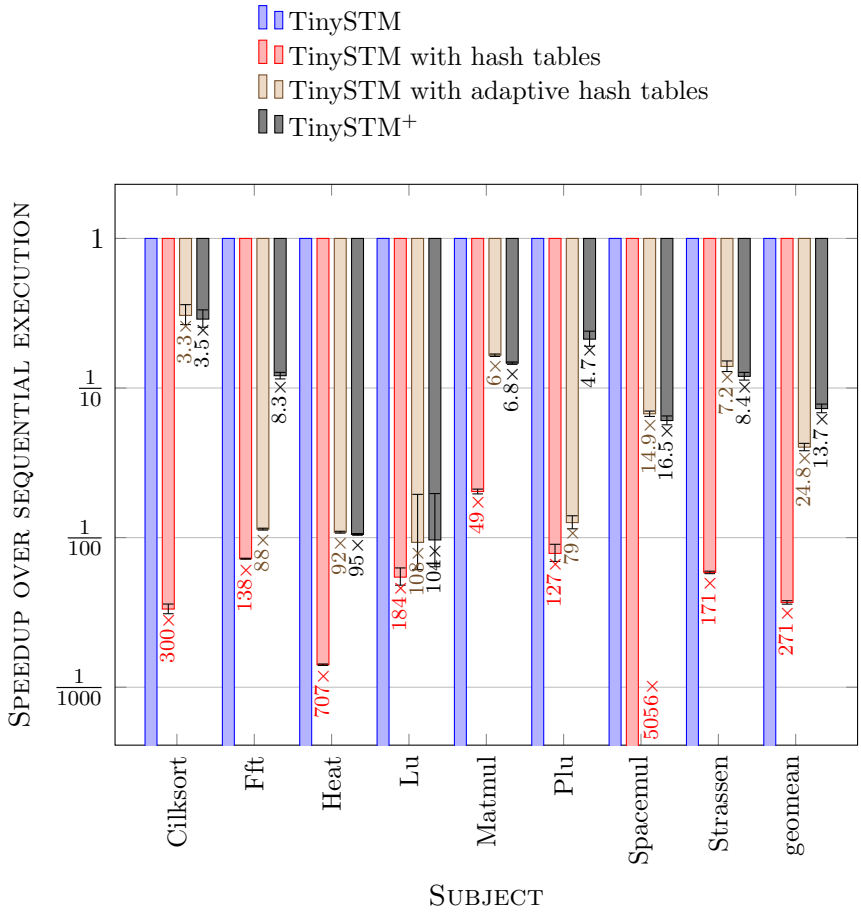


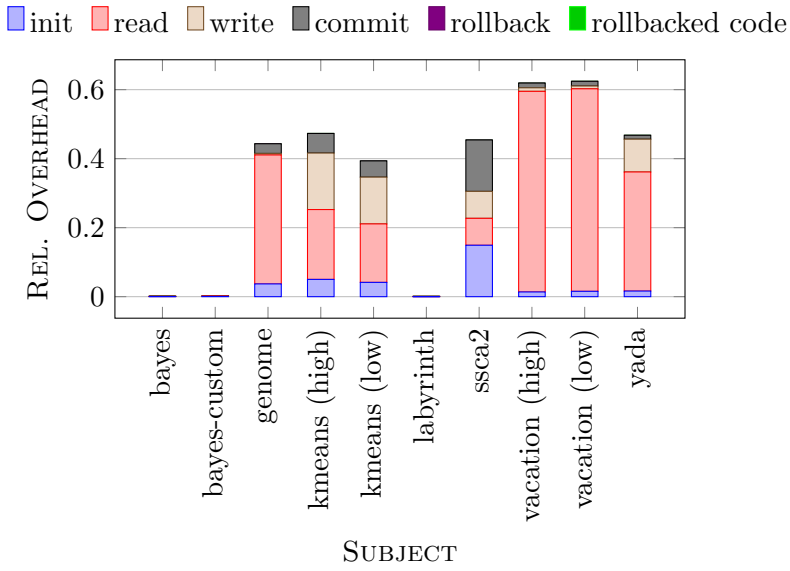
Figure 4.2: Runtime comparison of different data structures for read and write sets in STM, evaluated on automatically parallelized *Cilk* programs, executed in four threads. While the original TinySTM implementation (left bar) does not finish on any program within 12 hours, the addition of more sophisticated data structures reduces the overhead by orders of magnitude.

table sizes. The runtime of the full system including the hopscotch approach is shown by the gray bar. For some programs which greatly benefitted from adapting the initial hash table size, switching to hopscotch hashing results in a slight performance regression. The *Fft* and *Plu* programs on the other hand, which still trigger lots of hash collisions even with adaptive table sizes, greatly profit from the hopscotch hashing scheme. The geometric mean over all eight programs improves by another 45 % by using hopscotch hashing.

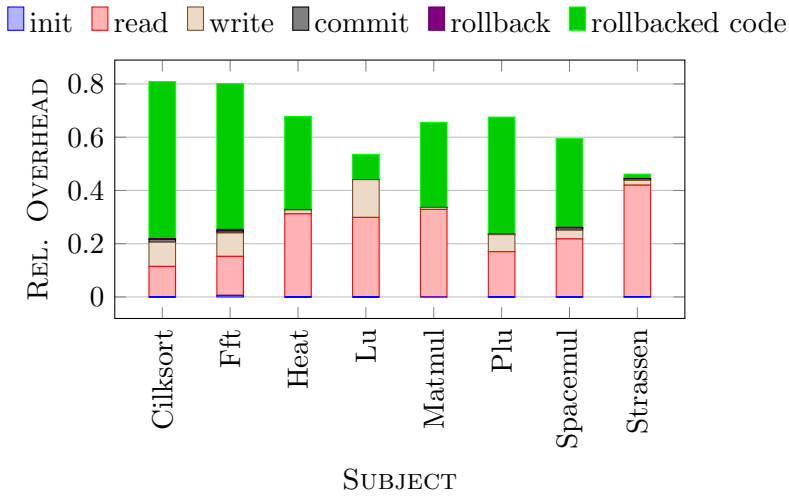
Now that we have seen that the redesigned data structures perform reasonably well on programs with large to huge memory footprints, we evaluate how this implementation performs on the STAMP programs. Table 4.2 showed that these programs often only read, and write to, a very small number of disjoint memory locations. For these cases the cost of a sophisticated hashing scheme may not pay off.

The performance breakdown we got on these programs using the modified TinySTM$^+$ are shown in Figure 4.3a. The overall distribution across the different STM operations is similar to the breakdown of the unmodified TinySTM shown in Figure 4.1a. For most programs the overall STM overhead is larger, but the difference is smaller than expected. These programs particularly profit from the adaptation of the initial size of the hash table. We always allocate a table with at least four entries, but even in the cases of just one write set entry, this small hash map does not seem to perform substantially worse than a simple array.

Figure 4.3b shows the corresponding breakdown for the cilk programs. The overall time spent in STM code is dramatically reduced in comparison to Figure 4.1b, where it was mostly reaching 100 %. The



(A) Overhead on the STAMP suite.



(B) Overhead on automatically parallelized *Cilk* programs.

Figure 4.3: Breakdown of TinySTM$^+$ overhead on different programs from STAMP and the *Cilk* example programs, executed in 4 threads. While the distribution on STAMP shows a slight regression compared to TinySTM (see Fig. 4.1a), the performance on the cilk programs is orders of magnitude better.

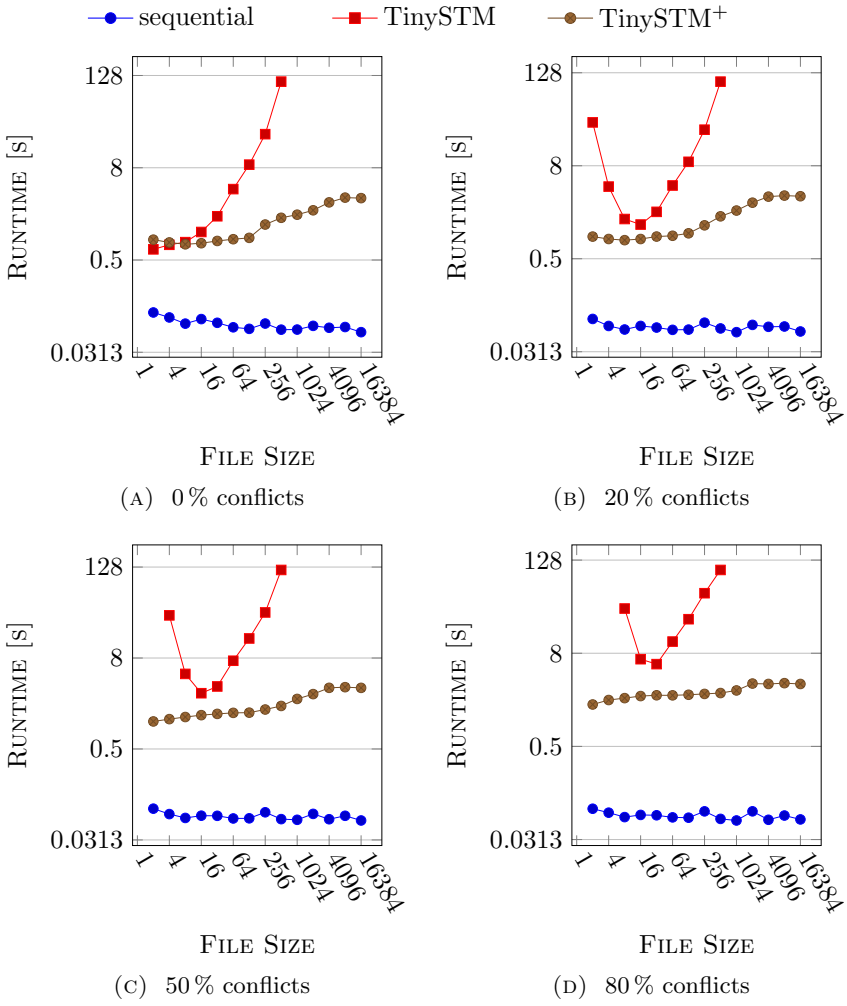
ratio between read and write overhead is about the same as before, but these operations are now performed much faster. The program hence reaches a point at which it detects misspeculation and performs rollbacks, making rollback cost visible more often.

4.3.3 Case study: Runtime Improvement in a Real-World Application

Beside the benchmark programs shown in the previous section, we also want to evaluate how the improved TinySTM implementation performs for parallelizing larger and modern C++ implementations.

In order to keep the environment controlled, we implement our own application from scratch: an indexer for C files, written in C++. It consists of a Lexer, returning the next token as a C++ object, and a Parser for a subset of the C language. The command line interface accepts a list of source files. It then processes all files in parallel, and puts all parsed definitions and usages of variables in a global database, implemented as a hash-table mapping symbols to lists of definitions. As data races might occur on this shared database, the parallel execution is wrapped in a transaction. All parallel code is instrumented offline, to avoid any influence of a runtime system. The source code for this case study is publicly available (see Section 7.2). It consists of 1,662 lines of code.

We execute this program called *cindex* with varying file sizes, but adapt the number of files such that the overall workload is always the same. The total number of lines in all files is always 2^{20}, and each line either uses or defines a variable. We vary the number of lines in each file between 2 and 2^{14}, hence the number of files varies



(A) 0 % conflicts (B) 20 % conflicts

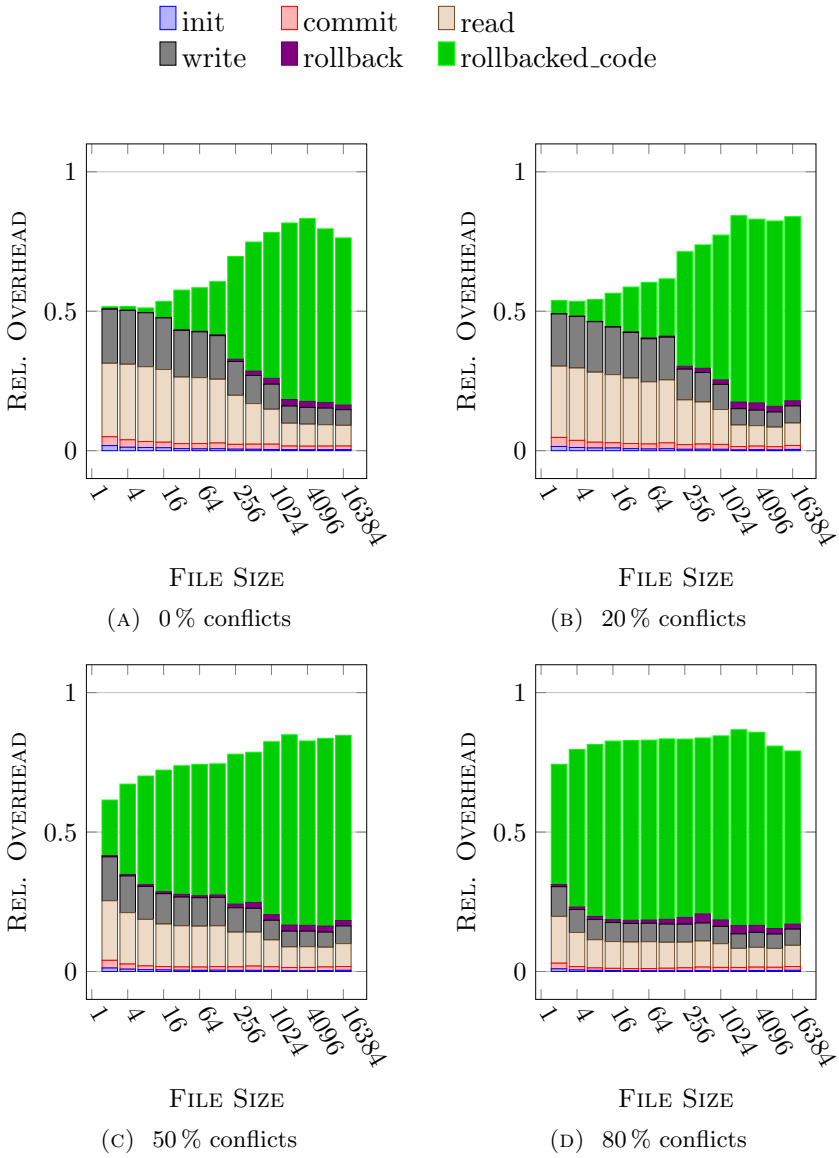
(C) 50 % conflicts (D) 80 % conflicts

FIGURE 4.4: Runtime for speculative indexing of C files of varying size and with a varying number of conflicts, executed either sequentially, or speculatively parallelized in TinySTM or TinySTM+. The overall workload is always the same. While the overhead of TinySTM grows quadratically with the file size, TinySTM+ shows a consistent overhead. Both, however, clearly fail to provide a speedup on four cores.

between 2^{19} and 2^6. We generate the files using a small python script which allows to set the number of conflicts between the files. In our experiment, we set the conflict rate to 0 %, 20 %, 50 %, or 80 %. We again execute each configuration ten times, and report the arithmetic mean over the first three quartiles, thus excluding the 25 % longest runs. The maximum sequential execution time over all configurations is 103 milliseconds, so we set the timeout for parallel execution to 210 seconds, which is more than 2000× the sequential execution time.

Figure 4.4 shows the results of this experiment. The first observation is that both TinySTM and TinySTM$^+$ take a multiple of the sequential execution time in all cases. Second, we see that while TinySTM$^+$ shows a consistent overhead independent of the size of the files, the overhead of the original TinySTM implementation increases quadratically with the file size, and always reaches the execution timeout for larger files. The overhead of TinySTM$^+$ over sequential execution varies between 9× and 56× for no conflicts, and between 22× and 56× for 80 % conflict rate.

In order to understand the causes of this large overhead, we again measured the relative execution time of individual STM overhead sources. This is plotted in Figure 4.5. We see that for low conflict rates and small file sizes about 50 % of execution time is spent in transactified code, meaning a slowdown of about 2×. However, for those small file sizes the overhead of thread management is the dominant factor, making parallelization non-profitable. For larger file sizes however, we quickly observe a large amount of rollback overhead, even if the actual conflict rate on the used and defined symbols is low. This is because of the hash table involved: Even though parallel tasks insert different symbols, they might end up in the same hash table



(A) 0 % conflicts (B) 20 % conflicts

(C) 50 % conflicts (D) 80 % conflicts

FIGURE 4.5: Overhead breakdown for the *cindex* program executed in TinySTM$^+$. File sizes again vary between 2 and 2^{14}, while the total workload is constant. We see that rollbacks quickly become the bottleneck in this benchmark.

bucket. The probability for this increases rapidly with the number of symbols added in each transaction, and quickly reaches nearly 100 % (similar to the well known *birthday paradox*).

4.4 Conclusion

In this chapter we have shown that using STM systems for speculative parallelization is not feasible in general. Even though the overhead of the most important STM functions can be improved substantially by using more scalable data structures, there is still a severe performance penalty for each single memory operation. This makes it hard to achieve any speedup with a number of processors as they are available in today's consumer hardware. STM in general can be sped up by using hardware extensions as they are available in many off-the-shelf processors nowadays. As discussed in Section 2.2.2, such extensions can not be used for implementing speculative parallelization though. The further part of this thesis will thus focus on systems that completely avoid the implicit tracking of each memory access.

Chapter 5

Virtual-Memory Based Speculation in User Space

A speculative runtime system for Thread-Level Speculation (TLS) can be implemented in many different ways, implicating different restrictions on the execution environment and inducing overhead at different points in time. The previous chapter evaluated the usage of Software Transaction Memory (STM) as a runtime system for TLS. STM tracks memory accesses explicitly by instrumenting the executed code such that memory operations do not directly operate on the main memory any more, but instead call a function of the STM runtime library to get the speculative value of a memory cell or store it respectively. As this imposes large overheads on the execution of speculative tasks, this chapter describes and evaluates an alternative technique which tracks memory changes without required instrumentation.

As discussed in Section 2.2.4, recent approaches for thread level speculation propose to make use of the *virtual memory system* implemented in the operating system and hardware architectures to isolate speculative from non-speculative state. This requires executing speculative tasks in separate processes instead of threads, thus turning multi-threaded into multi-process programs. By protecting the memory pages of the respective tasks and catching the resulting page faults, the runtime system observes which memory pages each process reads and writes. This information is used during commit to *validate* that the state produced by a speculative task is semantically correct and to *commit* its changes back to the main process.

This chapter describes a new runtime system based on these ideas, called U-TLS. It fully operates in user space, and communicates with the operating system via various system calls. U-TLS resembles the state of the art in virtual memory based TLS systems. Unfortunately, we could not reuse any existing approach, as none of them are publicly available. However, implementing U-TLS from scratch allowed us to explore different design decisions, and gave us valuable insights about the shortcomings of existing techniques.

5.1 Interface

This section describes the interface of U-TLS. The design goal is generalizability: The interface should be general enough to be used by all implementations of TLS which do not require code instrumentation. In particular, we use the very same interface also in our K-TLS approach described in the next chapter. Apart from this generalizability, the interface should also be easy to use by a developer as well

TABLE 5.1: TLS interface to be called from generated code.

function	parameters	description
tls_newList	–	allocate a new task list
tls_deleteList	*task_list*	deallocate a task list
tls_addTask	*task_list*, *input_size*, *output_size*, *func_ptr*	add a new task to the list, executing *func_ptr* with *input_size* bytes of input and *output_bytes* bytes of output
tls_getTasks	*task_list*	get a pointer to the start of the storage for this list
tls_getTasksEnd	*task_list*	get a pointer past the last byte of the storage for this list
utls_run	*task_list*	execute a task list in U-TLS

as by a parallelizing compiler. Hence it should consist of a handful of functions with clear semantics, such that the code to invoke these functions can easily be emitted or written by hand. The interface itself is implemented in C++, but C bindings exist for those functions which are invoked by the compiler. Table 5.1 shows the functions that are called by placing function calls in the generated code. Compared to the interface of an STM system (see Table 4.1 on page 63), we see that many fewer functions are involved, and they operate on a very different level than STM functions. This is because the U-TLS functions are not designed to be notified about each single memory operation of the program, but instead they set up individual tasks and manage their execution.

The central data structure of the interface is a *task list*. It contains all the tasks to be executed in parallel. Each task consists of a pointer pointing to the user code to be executed (*func_ptr*), some space for the input of the task and some space to write the output to. Pointers to the input and output space are passed to the user function, but only the second one may be written to. Hence the signature of the user function in C notation is `void (*fn)(const void*, void*)`. The order of the tasks in the list determines their commit order, hence it should match the sequential order of the tasks in the original program. The restriction to a list, implicating the requirement to establish a linear commit order, is a mere technical one. The extension to a directed acyclic commit graph, allowing for more flexible commit orders or even parallel commits, is left for future work. In our benchmarks it turned out that the tasks in most parallel sections are constrained to a linear commit order anyway, thus we would not profit from parallel commit or more relaxed commit orderings. The task list is passed to the respective runtime system for speculative parallel execution. In the case of U-TLS, this is the *utls_run* function. This invocation only returns once all tasks have executed and committed successfully. If there are tasks which fail repeatedly in speculative execution, e.g. because they abort the program, they are re-executed sequentially and non-speculatively by the runtime system.

The whole task list with all the input and output values is stored in one consecutive chunk of memory. This design is needed in the K-TLS implementation (see Chapter 6) in order to transfer the whole list to the kernel. U-TLS uses the C++ interface to the task list to access the individual tasks.

5.2 Design of U-TLS

In contrast to STM implementations, which fully operate in user-space, U-TLS communicates with the operating system via system calls. This removes the need to instrument the speculatively parallelized code. Because of this, it is possible to call external functions, e.g. from a pre-compiled library, within transactions. This not only lessens the requirements for code blocks to be parallelized, but also makes it easy to use for manual parallelization, because no compiler assistance is needed. U-TLS can be linked as an external library and be used via its interface as described above without taking any further steps. As U-TLS only handles memory accesses from within the speculative tasks, some restrictions have to be considered though, as detailed in Section 5.2.6.

The remainder of this section gives all the details of the design and implementation of the U-TLS system. The implementation is sketched in pseudo-code in Algorithms 5.1 and 5.2. Section 7.1 gives details about the concrete implementation and how to access it.

5.2.1 Data Structures

The sole input to the main routine of U-TLS is the task list as described in Section 5.1. The three pointers for the user function, input data and output data are named *funcPtr*, *input* and *output* in Algorithm 5.1. Additionally, U-TLS makes use of the following data structures: A global *TLSContext* holds information about the execution of the overall task list. In this simplified code this only consists of the set of pages modified by any committed task (*modified_pages*).

Algorithm 5.1 Pseudo-code implementation of U-TLS (first part; continued in Algorithm 5.2)

```
 1: procedure UTLSRUN(tasks)                                  ▷ main routine
 2:     N ← len(tasks)
 3:     ctx ← allocate TLSContext
 4:     states ← allocate shared TaskState[N]
 5:     for t ← 0 to N − 1 do
 6:         states[t].pipe ← PIPE( )
 7:         states[t].pid ← FORK(runTask, tasks[t], states[t])
 8:     for t ← 0 to N − 1 do
 9:         if not COMMIT(tasks[t], states[t], ctx) then
10:             for i ← t to N − 1 do
11:                 KILL(states[i].pid)
12:             for i ← t to N − 1 do
13:                 RUNUSERCODE(tasks[i])
14:             break

15: procedure RUNTASK(task, state)                            ▷ child process
16:     save state pointer into global variable (process-local)
17:     allocate a new stack and set RSP
18:     protect whole memory (except own stack)
19:     install segmentation fault handler SEGFAULT
20:     state.read_pages ← ∅
21:     state.modified_pages ← ∅
22:     RUNUSERCODE(task)
23:     state.finished ← true
24:     NOTIFY(state.ready)
25:     for each page_addr in state.modified_pages do
26:         WRITE(state.pipe, page_addr, PAGE_SIZE)

27: procedure RUNUSERCODE(task)          ▷ execute the code of one task
28:     functionPtr ← task.funcPtr
29:     functionPtr(task.input, task.output)

30: procedure SEGFAULT(page_addr)                             ▷ segfault handler
31:     if page_addr ∈ state.modified_pages then
32:         abort
33:     if page_addr ∈ state.read_pages then
34:         add page_addr to state.modified_pages
35:         grant write access to page (with COW)
36:     else
37:         add page_addr to state.read_pages
38:         grant read-only access to page
```

Algorithm 5.2 Pseudo-code implementation of U-TLS (second part; continuation of Algorithm 5.1)

```
39: procedure COMMIT(task, state, ctx)                    ▷ commit one task
40:     WAIT(state.ready or killed(state.pid))
41:     success ← state.finished and VALIDATE(state, ctx)
42:     if not success then
43:         state.finished ← false
44:         state.pid ← FORK(runTask, task, state)
45:         WAIT(state.ready or killed(state.pid))
46:         success ← state.finished
47:     if success then
48:         for each page_addr in state.modified_pages do
49:             READ(state.pipe, page_addr, PAGE_SIZE)
50:             add page_addr to ctx.modified_pages
51:     return success

52: procedure VALIDATE(state, ctx)                        ▷ conflict checking
53:     for each page_addr in state.read_pages do
54:         if page_addr ∈ ctx.modified_pages then
55:             return false
56:     return true
```

It is updated during commit and used for validating later tasks. The second structure allocated is an array of *TaskState*s. Each *TaskState* holds the following fields:

- ***pipe*** to hold the file descriptor of a unidirectional *pipe* for communicating modified pages from the child task to the parent.
- ***pid*** the process id of the forked child process.
- ***read_pages*** the set of pages read by this task.
- ***modified_pages*** the set of pages modified by this task (always a subset of *read_pages*).

- ***finished*** a flag to indicate whether the task successfully finished the execution of the user code (false indicates premature termination, e.g. because of a signal).

- ***ready*** a condition variable to communicate to the parent that the execution of user code finished.

5.2.2 Forking Speculative Tasks

In order to execute a task list in U-TLS, the developer or the automatic parallelizer invokes the UTLSRUN routine. Before forking the actual processes to execute the user code, U-TLS allocates a *TLSContext* (line 3) to hold the set of pages modified since the process forked (*ctx.modified_pages*). Also, for each task a *TaskState* structure is allocated to hold information about the respective task with the fields as described before. Since some of this information is updated by the child, but read by the parent, we allocate the *TaskState* structures in memory shared between the main process and its children. The parent also opens a unidirectional communication channel (a *pipe*) per process (line 6) to transfer back changed memory pages in the commit phase.

After this setup, the actual child processes are forked to execute the RUNTASK routine (line 7), and then committed in order (lines 8 to 14, cf. Section 5.2.4). If this commit fails repeatedly for any of the tasks, e.g. because the task was killed by a signal, then the processes executing the remaining tasks are killed, and the respective code is re-executed in the main process sequentially (lines 10 to 13).

5.2.3 Execution of a Speculative Task

Each spawned process starts execution in the runTask routine. Before the actual user code is executed, the child process needs to be set up properly (lines 16 to 21). First, the *state* pointer, which is passed from the parent, is saved to a global variable, such that it is available to the segmentation fault handler. Note that this change of the global variable is only visible to this specific child, since the operating system automatically creates private copies of all changed memory pages. As this page is written before any pages are protected, this change will not cause memory conflicts and the respective page will not be copied to the parent because of this write. Next, a new memory region for the stack is allocated, such that stack operations of the different child processes do not collide. By setting the stack pointer (*RSP*) to the top of this new region, the user code will allocate all new stack frames there.

Apart from the newly allocated stack and the *TaskState* structures, all writable memory regions are made inaccessible by *mprotect* system calls. The memory regions of the process are determined dynamically from the virtual `/proc/self/maps` file. This ensures that a *segmentation fault (segfault)* is triggered whenever the user code tries to access (read or write) any memory in these regions. This segfault is handled by a custom segfault handler (lines 30 to 38), which records that the page was accessed by the process, and makes it available read-only. On the second segfault per page, we know that this must be a writing access, since reads were already allowed. Hence, also write access is granted, and the page is stored in the set of modified pages. If a third segmentation fault happens, this can only mean that the previous

mprotect calls did not succeed, hence the memory address is illegal. This happens for example when a task accesses memory through a pointer which should have been updated by a previous task, but this update is not visible to the process. In this case, we just abort the execution of the process, and the main process retries execution later, when all previous tasks have committed (see Section 5.2.4). In fact, we check the return code of the previous *mprotect* system calls directly, in order to detect such situations already on the first segmentation fault.

After returning from the actual user code of the respective task (line 22), the *finished* flag in the *TaskState* is set to signal that the task terminated regularly. Then, the parent is notified of the completion of the process, and all modified memory pages are transmitted to the parent process via the pipe established before forking (line 26). If available, we use the *vmsplice* system call for this, which has potentially better performance than just writing the data page by page.

5.2.4 Validating and Committing Speculative State

Before starting the actual commit phase, the parent process first waits until the child process either signals that it finished execution of the task code via the *state.ready* condition, or the task exits prematurely (line 40). The latter might happen if the code tries to access inaccessible memory, e.g. via an invalid pointer, or because the program aborts explicitly, e.g. via an assertion. In this case, *state.finished* is still *false*, and the task is considered failed. Otherwise, validation is performed (line 41) by checking for an intersection

between all pages read by the child process (*state.read_pages*) and all pages modified since forking it (*ctx.modified_pages*). If there is an intersection, the child process might have read outdated data, and is also considered failed.

If any of these two checks fail, the task needs to re-execute (lines 43 to 46). This new fork will now see all memory updates by previous tasks, and thus no validation needs to be performed afterwards, as there cannot be any read-after-write conflicts. If this new process still does not execute the user code without aborting in between, the commit is aborted. Subsequently, it will be executed in the main process directly (line 13), such that any signals will be delivered to the main process.

Finally, if either the first execution or the re-execution succeeded, the actual commit is performed (lines 48 to 50). The content of all modified pages is read from the pipe connecting the two processes, and written to the corresponding location in the non-speculative memory. All modified pages are registered in the *ctx.modified_pages* set for validation of subsequent tasks.

5.2.5 Optimizations

Since the first task in each task list can never conflict with any other task, we do not need to track the pages read by this process. Hence, all memory is initially read-only instead of inaccessible for the first task, and the segfault handler immediately grants write access. The same reasoning applies for re-executed tasks: Since they are spawned only when all preceding tasks have already committed, they require no validation, hence no read set needs to be tracked. This optimization

saves a lot of unnecessary context switches due to page faults induced by read accesses.

Also, if there are more tasks than hardware threads available in the system, it makes sense to only spawn as many tasks initially as there are hardware threads, and spawn the next task whenever a task finishes. This would call for a more sophisticated verification scheme: Instead of just memorizing which pages have been modified, a global clock (or *version number*) can be associated with each page, tracking which task modified the page last. When forking a new process, the *version* of the global memory (i.e. the sequence number of the last committed task) can be stored, and a memory conflict is only reported if at commit time any read page has a version number greater than this stored version. Thus a conflict is only detected if the process actually used an outdated memory page. This concept is similar to *time-based STM systems* [24, 89]. We did not implement this optimization in U-TLS yet, since our benchmarks do not spawn more tasks than the number of available hardware threads. Most automatic parallelization systems follow this principle, so the value of adding the optimization to U-TLS is questionable, and would not be observable in our evaluation. We thus leave it for future work.

5.2.6 Restrictions of U-TLS

Obviously, the U-TLS system can only be used on operating systems offering the facilities used in the implementation. These include in particular copy-on-write process forking, protecting individual memory pages for either read-only or no access, customized segmentation

fault handlers, and inter-process communication to copy back changed data. All POSIX compliant systems provide these functions.

Apart from that, there are also restrictions on the executed user code. Since conflict checking and commit only handles memory effects, there should be no other side effects within a task. Uncaught side effects include any file operations, like opening or closing file handles, or reading or writing to them. Those effects will neither be applied in order, nor can they be rolled back. Even though the parent can be protected from damage by these side effects by not inheriting the file descriptor table, but creating a copy for the child, this still does not guarantee to preserve the semantics of sequential execution. Other side effects, like creating new memory mappings, (un-)protecting memory regions, any file system operation or other externally visible effects cannot be prevented by this approach either. Depending on the type of the operation, its effect will either not be applied to the main process (e.g. for *mmap* or *sigaction* calls), or the order of the operations might be different than in sequential execution (e.g. for file operations). Because those changes are not included in the rollback, they might even be applied a second or third time during re-execution.

Also, care has to be taken if the original application is already multi-threaded. As U-TLS only handles data dependencies between the speculative tasks, data races with concurrently executing threads may still occur. Since U-TLS exchanges whole memory pages during commit, new data races may even be introduced by overwriting unrelated sections within pages. Hence we only evaluate U-TLS and all other TLS systems on single-threaded applications. This is a common requirement for automatic parallelization approaches.

5.3 Evaluation

In order to evaluate the performance of the U-TLS system, we run micro-benchmarks to assess aspects of U-TLS itself, and execute parallelized programs with both U-TLS and our improved STM system called TinySTM$^+$ (see Section 4). The system used for this evaluation is equipped with a quad-core Intel i7 870 CPU running at 2.93 GHz and 16 GB of main memory. We execute each benchmark at least ten times and report the arithmetic mean over the first three quartiles. This excludes runs that were unexpectedly interrupted by unrelated events or processes running concurrently on the same machine.

5.3.1 TLS Overhead

The first part of the evaluation measures the overhead of the primitive operations of a TLS system: the time for spawning tasks, the overhead that the speculation system induces during the parallel execution of the tasks, and the validation and commit time. In order to measure those numbers, we run some micro-benchmarks as described in the following sections.

5.3.1.1 Spawning Tasks

Most runtime systems that track memory accesses explicitly (like STM) execute parallel tasks in individual threads. U-TLS needs different virtual memory mappings for each task and therefore has to

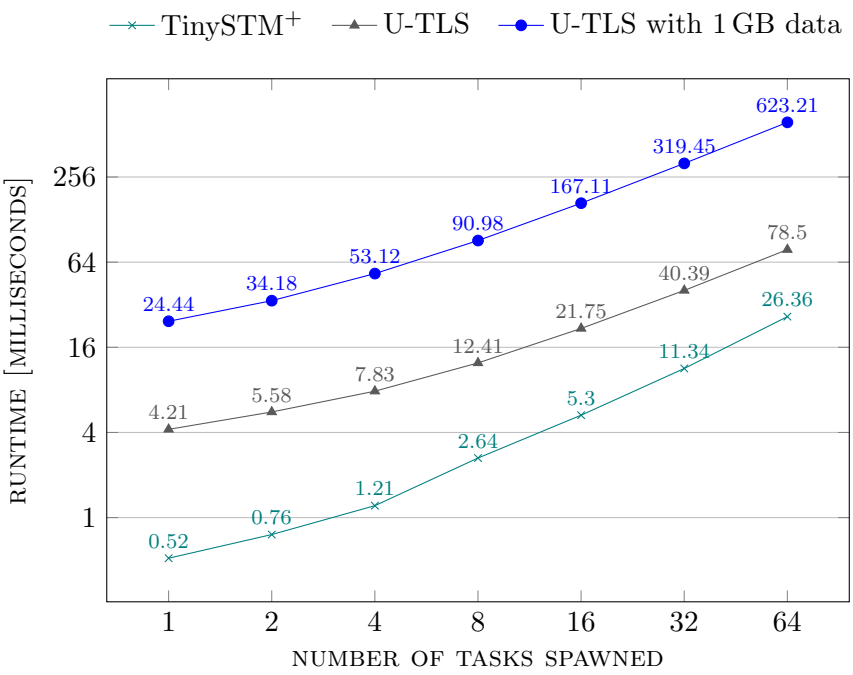


FIGURE 5.1: Overhead of spawning a task list of varying size in U-TLS against TinySTM+. Since TinySTM+ just as most STM systems uses threads instead of processes, it spawns tasks the fastest (around 0.3 ms per task). U-TLS uses a *fork* system call per task, which takes around 1.2 ms, plus initial 4 ms per task list. The time increases further if more page table entries have to be copied.

fork individual processes. Hence, its initial overhead is larger, whereas the overhead during task execution is potentially much smaller.

In this benchmark, each run creates a task list of N tasks, where N varies between 1 and 64. We measure the overall wall-clock execution time of executing this task list for both TinySTM+ and U-TLS. We execute two benchmarks per system. First, we spawn the tasks immediately after starting the application, i.e. without having performed any work yet. In the second benchmark, we allocate 1 GB

of memory and initialize it to zero. This causes physical pages to be allocated for this memory and the page table to be filled with these pages. In all cases, the time for validating and committing the empty transactions is negligible; the times reported are indeed caused by spawning the threads or processes, and protecting the memory. Many numbers are in the range of milliseconds. Therefore, we cross-validated the experiment with 100 and 1000 iterations and validated that the measurements are reliable.

Figure 5.1 shows the result of this benchmark. As expected, forking processes takes considerably longer than spawning threads, as the operating system has to clone more resources like the signal table, the page table and other internal data structures. STM takes around 0.2 ms per task if the number of threads spawned is below the number of cores, and up to 0.5 ms otherwise. As this number is independent of the allocated memory, we only plot one line for TinySTM$^+$. Spawning a single task without much memory consumption in U-TLS via the *fork* system call takes about 4 ms, and 1.2 ms for each additional one. Additional tasks cause less overhead than the first task, since some setup work of the tasks—like protecting writable memory via *mprotect* calls—is executed in parallel by all tasks. When additional 1 GB of memory were allocated before the *fork*, the time for the first task increases to 24 ms, and 9.5 ms are spent for each additional task. This additional time is spent in iterating over the page table for the additionally allocated memory range, creating the respective page table entries in the forked process, making the respective pages shared between the processes by removing write access in the parent process, and for clearing the page table entries again when the memory range is *mprotect*ed during the setup of the child process.

5.3.1.2 Execution Overhead

The second type of overhead happens during execution of the actual task in order to track the memory accesses during runtime. In STM, this is done explicitly in software by keeping a read and a write set which is inspected and updated during transactional load and store operations (cf. Section 4.3.1.1). In U-TLS, the overhead is mainly caused by two actions: the context switches between user space and kernel space for each page fault, and creating private copies of pages which are modified by the transaction. The number of context switches is quite large. Whenever a page is accessed for the first time, or first accessed via a writing operation, one to two page faults are triggered, leading to six to eight context switches as illustrated in Figure 5.2:

1. from user space to kernel space for handling the page fault, triggered by the MMU, part of the CPU;

2. from kernel space to user space for handling the segmentation fault triggered by the page fault handler because the access right of the respective page does not allow for the given memory access;

3. from user space to kernel space for executing the *mprotect* system call to allow the respective memory operation on that page;

4. back to user space when returning from the *mprotect*;

5. back to kernel space when returning from the segfault handler, telling the operating system to retry execution of the instruction which triggered the segmentation fault;

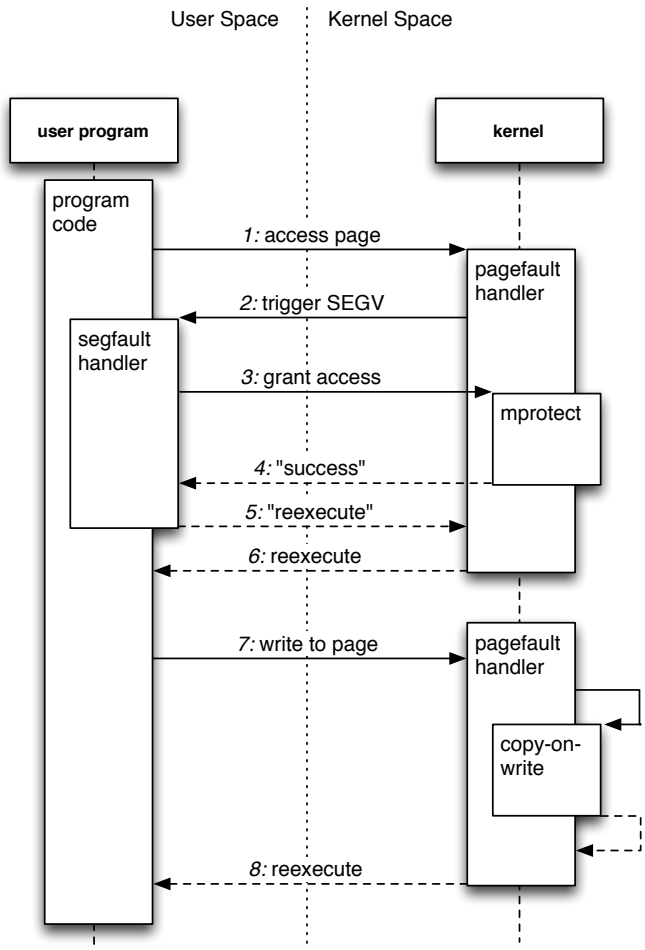


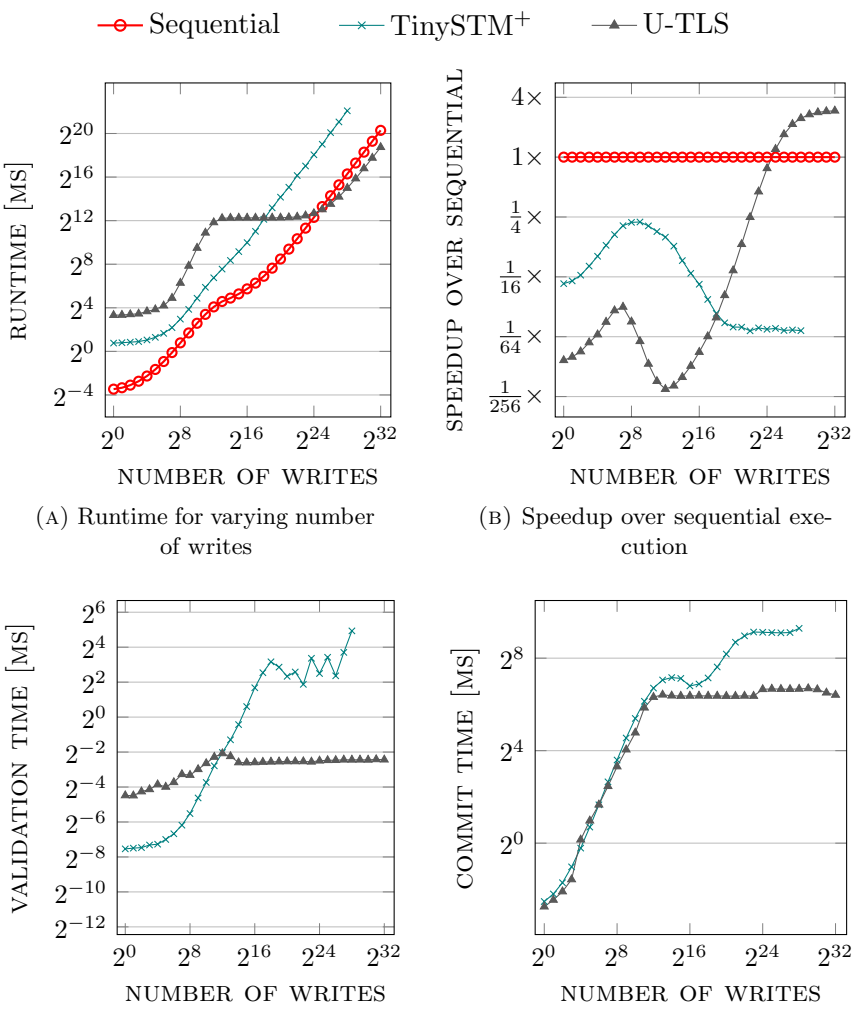
FIGURE 5.2: Handling of page faults during execution in U-TLS. On the first access to a page, the page fault handler will be invoked, leading to six context switches (1–6). On a later writing access, two additional page faults will be triggered, leading to another eight context switches (1–8). In total, 14 context switches are required for each modified page.

6. back to user space for reexecuting the memory operation;
7. *(only for writing accesses)* again from user space to kernel space for handling the second page fault on the same page;
8. *(only for writing accesses)* back from kernel space to user space after creating a private copy of the previously shared page.

The first access to a page will always only make it accessible read-only, thus executing only the first six steps. If the page is later accessed by a writing memory operation, another two page faults will be triggered, leading to all eight steps being executed. Thus, a writing access to a previously untouched memory page leads to a total of 14 context switches to make the page accessible for writes. Later reads or writes, however, do not trigger any page faults any more.

In order to compare this runtime overhead of U-TLS against that of TinySTM$^+$, we run a benchmark in which four parallel tasks perform random write accesses to disjoint memory blocks. The memory area which is updated by each task has a size of 16 MB. Figure 5.3a shows the runtime of both systems plus the sequential execution with respect to a varying number of memory accesses. Figure 5.3b shows the corresponding speedup over sequential execution. Note that all numbers are reported on a logarithmic scale on both axes.

In this benchmark TinySTM$^+$ never succeeds to beat the sequential runtime. This is mainly because of the huge overhead caused by the explicit tracking of memory accesses (see Section 4.3), but also because of the rollbacks it performs. TinySTM maps written memory addresses to a *lock array* of fixed size, so there are hash collisions which provoke false rollbacks. From 2^{16} on TinySTM$^+$ executes



(A) Runtime for varying number of writes

(B) Speedup over sequential execution

(C) Time spent for conflict checking

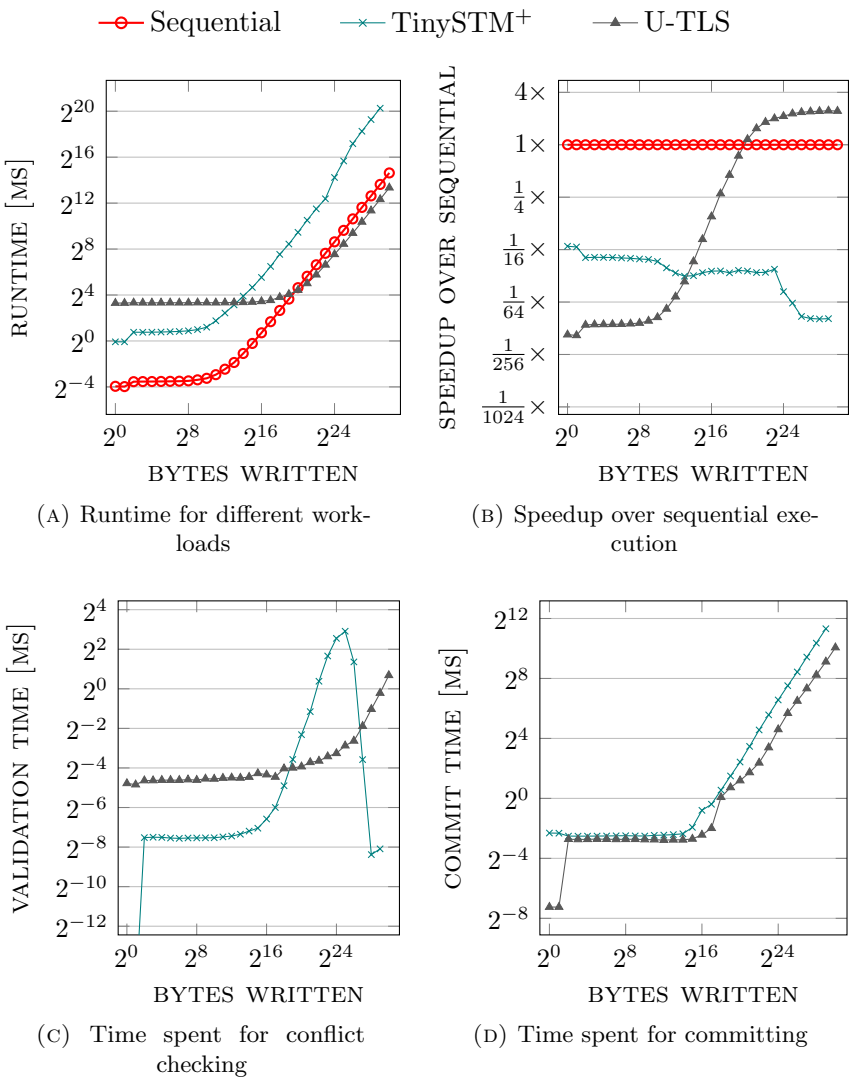
(D) Time spent for committing

FIGURE 5.3: Performance of TinySTM+ and U-TLS in an artificial benchmark, in which each task randomly updates memory cells within a 16 MB memory block. For a small number of memory accesses, the STM system performs best, but still falls behind sequential execution. U-TLS shows problems in the mid-range, but performs significantly better than TinySTM+ for large workloads. It reaches a speedup over sequential execution of about 3× for large workloads.

on average more than one rollback per execution of four tasks and reaches a 50 % rollback rate for more writes.

U-TLS starts with moderately more overhead than TinySTM$^+$ for small workloads. This is expected since for the random memory accesses many pages are copied just for a few memory updates performed on each page. Also, the setup cost per task are larger. For a mid-sized number of writes U-TLS shows surprisingly large overheads. Profiling reveals that the repeated change of access rights on individual memory pages fragments the virtual memory descriptor in the kernel, and increases the number of *virtual memory areas (VMAs)* up to several thousand. The Linux kernel organizes the virtual memory descriptor as a linked list, with an additional red-black tree for faster lookup. Because it is traversed on each page fault (and on other operations), this leads to a severe slowdown in the kernel code. As more and more pages get unprotected, the respective memory areas are merged again, mitigating this slowdown for larger memory footprints.

In order to validate this hypothesis, we ran another benchmark in which each transaction writes linearly to a disjoint memory block. The results are shown in Figure 5.4. In this test, the sequential execution takes considerably shorter time (especially for large inputs), since the cache utilization is much better. TinySTM$^+$ and U-TLS still have their constant overhead per transaction, thus showing slowdowns again for very small tasks. TinySTM$^+$ again stays between 14× and 100× slowdown for all inputs. Now in this benchmark—as projected—U-TLS does not show the super-linear slowdown for mid-sized writes, as the fragmentation of the virtual address space does not occur here. Instead, the execution time stays flat up to about 2^{16} writes, since up to this point the initial overhead dominates the runtime



(A) Runtime for different workloads

(B) Speedup over sequential execution

(C) Time spent for conflict checking

(D) Time spent for committing

FIGURE 5.4: Second benchmark comparing U-TLS performance against TinySTM+. This time each task writes a linear block of memory with varying size. All blocks are on disjoint memory pages. Again, parallelized execution with TinySTM+ takes longer than sequential execution in all cases, while U-TLS reaches 2.6× speedup for a large number of writes. In contrast to Figure 5.3, the performance of U-TLS increases monotonously with the workload.

overhead for handling the page faults. The overall speedup for large workloads only approaches 2.6× in this benchmark. This is caused by the different access pattern. Each memory cell is only written once, in contrast to the multiple updates of the same memory in the previous benchmark. Because of this different pattern, the cost for copying and later committing a page amortizes less over the execution time. You can also see that the commit time for this benchmark continues rising for larger input, where it stayed flat before as soon as all memory pages were written at least once. But interestingly, in this benchmark U-TLS shows speedup over sequential execution already for smaller inputs—on 2^{20} instead of 2^{25}.

5.3.1.3 Validation and Commit

After parallel execution, U-TLS validates the set of read and written memory pages and then commits them by writing them to a pipe connecting the child process to the parent process. TinySTM$^+$ on the other hand validates and commits memory changes at the machine word granularity. Figures 5.3c and 5.4c show the absolute time taken for validation of all tasks. In TinySTM$^+$ validation and commit can happen in parallel in the individual transactions. For the reported number, we sum up the wall clock time in all parallel threads, thus the number could even exceed the total runtime reported—which it does not do in our benchmarks. In U-TLS we measure the validation time as the time it takes the parent process to receive the information about which pages were read and written by the respective task and check them against the set of pages modified by previous tasks. For the random writes within a 16 MB block of memory, the validation time for U-TLS is negligible in all cases. For the linear write to

memory it only crosses the 1 ms boundary for very large workloads, and compared to the overall execution time it is also negligible for all inputs. For TinySTM$^+$ it also never exceeds 1 % of the total execution time, but grows much higher than for U-TLS, since each single written memory address must be validated using atomic memory operations. For more than 2^{24} written bytes, the validation time drops by orders of magnitude, since the falsely detected memory conflicts occur much earlier during the validation.

The commit time is plotted in Figures 5.3d and 5.4d. In this benchmark, the commit times measured for TinySTM$^+$ exceed the total execution time for the random updates in the range from 2^6 to 2^{11}, suggesting that a major part of the execution time is actually spent for committing. For U-TLS we measure the time it takes the parent process to receive the content of all modified pages and write them to its own address space. This commit time is much larger than the validation time, and reaches a maximum of 16.9 % of the total execution time for the linear memory updates. For the random writes, it increases until about 4096 memory accesses, which is the point where most memory pages in the 16 MB range have been touched at least once. There it contributes 4.9 % of the total execution time. From this point on, the commit time stabilizes since fewer pages are added when increasing the number of accesses. It even slightly decreases, since a more continuous block of memory can be transferred through the pipe. TinySTM$^+$ takes not much longer to commit for a small number of updates, but then the gap increases as more bytes per page are written. It increases until also most individual words have been written at least once. For the benchmark executing linear memory accesses, the commit time for U-TLS stays flat until a full page has

been written (at 2^{12} bytes), then it increases linearly. TinySTM$^+$ again consistently takes more time to commit—by more than a factor of four for large workloads, and 15 – 30 % more for small inputs.

5.3.2 Usage in Automatic Parallelization

The previous section has shown that for large transactions, utilizing the virtual memory system provides a much better performance than tracking memory explicitly. This section evaluates how these performance benefits translate into speedup of speculatively parallelized real-world programs.

While the previous benchmarks were statically compiled programs, this time we execute the programs in *Sambamba* (see Chapter 3) for automatic parallelization. The benchmarks we have chosen for this evaluation are the *serial elision* of the *Cilk* [9] program suite. The serial elision of a (manually parallelized) Cilk program can be generated easily by ignoring all *spawn* and *sync* keywords. This suite contains mostly programs working in a divide-and-conquer manner, writing the computed results in a shared array or matrix. This shared object often causes false data dependencies to be detected statically, because state-of-the-art alias analyses cannot proof the accesses disjoint. Additionally, there are real data dependencies caused by memory allocation, accesses to shared objects on the heap, or premature termination via assertions or explicit aborts. Hence speculation is needed in order to parallelize those programs automatically.

We briefly introduce the eight programs from the *Cilk* suite which were automatically speculatively parallelized by *Sambamba*. The input

parameters used for the evaluation are listed together with more statistical data and results of both U-TLS and K-TLS in Table 6.1 on page 150. Several programs from the Cilk suite were excluded either because they do not operate on shared data (and therefore need no TLS), use explicit locking, or use Cilk intrinsics like *inlets* for which there exists no serial elision. The following programs were used for this evaluation:

Cilksort a sorting algorithm which uses mergesort with parallel sorting and parallel merging, and switches to quicksort for smaller arrays.

Fft an implementation of fast fourier transform.

Heat simulates heat diffusion by running a number of Jacobi iterations. The rows of the grid which is transformed in each iteration are allocated on the heap and accessed via two levels on indirection.

Lu a naive implementation of LU decomposition, which factors a matrix as the product of a lower triangular matrix and an upper triangular matrix.

Matmul which implements the multiplication of two rectangular matrices by divide and conquer.

Plu another implementation of LU decomposition with partial pivoting.

Spacemul an optimized implementation for matrix multiplication of square matrices.

Strassen which implements the Strassen algorithm for multiplying square matrices.

Even though there are three implementations of matrix multiplication, they show very different memory access patterns. *Matmul* recursively splits the matrix along the largest dimension, which leads to striped memory accesses if the largest dimension is not the x dimension. *Spacemul* splits the square matrix in four quarters in each recursion step, providing more parallelization opportunities. *Strassen* also splits the matrix into four quarters, but processes them in a different order, leading to less consecutive accesses.

For all programs we generated the *serial elision*, which resembles a correct sequential execution of the program. We then placed manual parallelization hints for speculation because the parallelization analysis of Sambamba is not tailored towards speculation yet [112]. We changed the memory allocation in all programs such that large objects are automatically aligned to 4096 bytes (the size of a memory page). This ensures that partitions of the data by a power of two are likely to reside on separate pages. This transformation could also be fully automated by a parallelizing compiler by installing a custom memory allocator or transforming the relevant memory allocation sites.

In each program between one and four locations are parallelized. The locations are always at the kernel of the computation, which is either a recursive function or a loop. Speculation is often only needed because of the imprecision of static analyses. Experts can reimplement these algorithms without the need for speculation. Automatic approaches however can not.

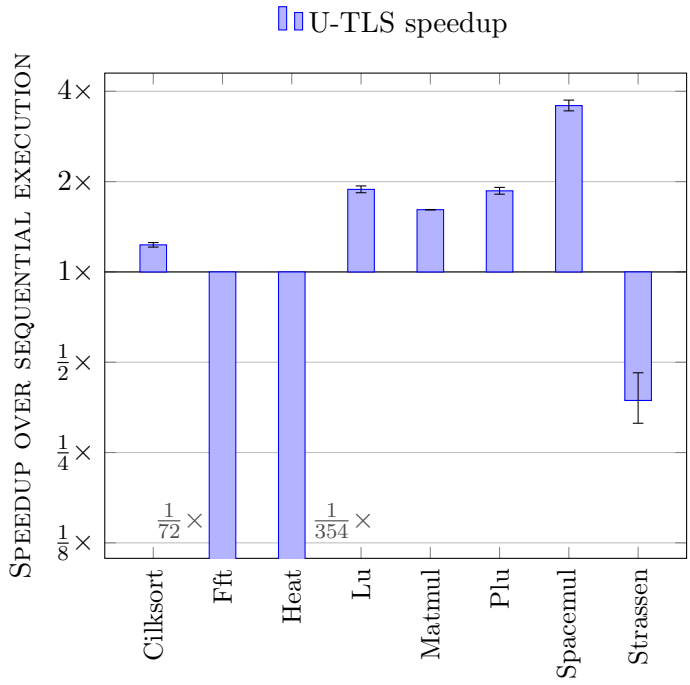


FIGURE 5.5: Speedup of U-TLS achieved by automatic parallelization of eight programs from the *Cilk* suite.

Figure 5.5 plots the speedups measured on the eight programs. We observe that U-TLS is able to speed up five of the programs, but only provides really good performance on the *Spacemul* program. While the slowdown on *Strassen* is moderate, for *Fft* and *Heat* it is severe. The geometric mean of the speedup of U-TLS is $\frac{1}{2.69}\times$, which translates to a 2.69× slowdown. Without the *Fft* and *Heat* programs it would be a 1.45× speedup. If we compare this to the numbers generated with STM (see Section 4.3.2), we see the huge advantage of U-TLS over TinySTM$^+$: The latter had a 13.7× slowdown for the same set of programs. Interestingly, though, for *Fft* and *Heat*, our outliers in the above evaluation, TinySTM$^+$ performed better.

The large slowdowns of U-TLS are often caused by scattered memory accesses, which cause fragmentation in the virtual memory descriptor of the process (see Section 5.3.1.2). On *Fft* and *Strassen* this is caused by the accesses to the shared array. *Heat* allocates the rows of the grid as individual objects on the heap, so they are not consecutive either. A further description of the characteristics of the different programs and explanations for the performance observations are given in Section 6.3.2.

5.4 Conclusion

In this chapter we presented the design and implementation of a *virtual memory based* system for *thread level speculation*, called U-TLS. Even though others proposed similar systems before, none is available for evaluation. By using a very simple interface for creating task lists and executing them in U-TLS, we allow for an easy use in manual as well as automatic parallelization. The implementation is independent from any embedder and even from the source language. It can be compiled and executed on any POSIX compliant operating system and consists of less than one thousand lines of code.

The evaluation shows that especially for large tasks which are typically aimed for by automatic parallelization, U-TLS is indeed able to provide a speedup close to full parallel execution without any runtime system. The STM system as evaluated in Section 4.3 failed to provide any speedup for those programs. However, U-TLS has a substantial overhead for spawning the processes to execute the speculative tasks. For small tasks, it therefore shows slowdowns that are similar to execution in TinySTM$^+$. Also, if the memory accesses executed by

the speculative task are random within a large memory block, it fragments the virtual address space, leading to large overheads for handling further page faults in the kernel.

We conclude that utilizing the *virtual memory system* for TLS is a promising approach, but certain operations need to be executed more efficiently. We will try to achieve this by implementing major parts of the TLS directly in the operating system, as described in the next chapter.

Chapter 6

Virtual-Memory Based Speculation in Kernel Space

As shown in the previous chapter, U-TLS provides much better performance than STM on typical automatically parallelized programs. It still shows considerable overhead for all relevant phases of speculative execution though: forking the processes to execute speculative tasks, executing the user code itself, and writing back changes to the main process. All of those operations can be sped up substantially by not implementing them via system calls, but directly in the operating system. Additionally, speculative tasks can be fully isolated from each other even in the presence of arbitrary system calls inside the speculative code.

This chapter describes the design, implementation and evaluation of this novel approach called K-TLS.

6.1 Design of K-TLS

As K-TLS executes major parts of the orchestration of speculative tasks directly in the operating system, its implementation is split in two parts: A Linux kernel module for the low-level process and memory management, and a user-space library which communicates with the kernel module and handles sequential re-execution appropriately.

As the kernel module is able to intercept any system call executed from within speculative tasks, it can effectively prohibit any action which is not covered by the memory protection mechanisms involved. This includes not only file I/O, but also sending signals to other processes, starting new programs, forking a process, changing signal handlers, starting timers, and many more.

K-TLS therefore poses no restrictions on the code to be executed within speculative tasks, but guarantees sequential semantics in any case. In uncertainly, K-TLS will conservatively abort a task and trigger sequential re-execution.

This section describes the design and implementation of both the kernel module as well as the user-space interface as shown in Algorithm 6.1.

6.1.1 Data Structures

Similar to the U-TLS design, K-TLS also allocates one data structure (*KTLSContext*) holding information about the execution of the whole task list, and one structure (*TaskState*) per task. *KTLSContext* consists of

- ***parent_task*** a pointer to the kernel structure describing the parent process (the one which issued the *ioctl* system call to start speculative execution).

- ***version*** an integer value containing the *version* of the main memory relative to the start of the whole task list. It is initialized to zero and incremented each time a task commits.

- ***page_versions*** a map containing the *version* of each single page in memory. This map is updated during commit and used to detect conflicts between speculative tasks (see Section 6.1.5).

A *TaskState* holds information about the execution of one single speculative task:

- ***ctx*** a reference to the *KTLSContext* structure wrapping the execution of the whole task list.

- ***start_version*** the version of committed state in main memory when the first page fault happened.

- ***stack*** a pointer to the bottom of the memory region used for the stack of this speculative task.

- ***proc*** a pointer to the memory structure describing the *process* (also called *task* in Linux) which executes this speculative task.

- ***touched_pages*** a set of all pages which were accessed during the execution of this task.

Algorithm 6.1 Pseudo-code implementation of K-TLS

```
 1: procedure runTasks(tasks)                                  ▷ user-space interface
 2:     fd ← open("/dev/ktls")
 3:     numExec ← ioctl(fd, KTLS_RUN, tasks)
 4:     for i ← numExec to len(tasks) − 1 do
 5:         functionPtr ← tasks[i].funcPtr
 6:         functionPtr(tasks[i].input, tasks[i].output)

 7: procedure ktlsRun(tasks)                                   ▷ kernel-space entry
 8:     N ← len(tasks)
 9:     ctx ← allocate KTLSContext
10:     ctx.parent_task ← current
11:     ctx.version ← 0
12:     ctx.page_versions ← allocate hash map
13:     states[] ← allocate N ∗ TaskState
14:     for i ← 0 to N − 1 do
15:         states[i].ctx ← ctx
16:         spawnTask(tasks[i], states[i])
17:     exec ← 0
18:     while exec < N do
19:         waitForCompletion(states[exec].task)
20:         valid ← validate(ctx, states[exec])
21:         if not valid then
22:             spawnTask(tasks[exec], states[exec])
23:             waitForCompletion(states[exec].task)
24:         if states[exec].task.exit_code ≠ 0 then
25:             break
26:         commit(ctx, states[exec])
27:         exec ← exec + 1
28:     for i ← exec to N − 1 do
29:         kill(states[i].task)
30:     return exec

31: procedure spawnTask(task, state)                           ▷ spawn new task
32:     state.proc ← copyProcess(current)
33:     protectMemory(state.proc)
34:     state.touched_pages ← allocate hash set
35:     state.stack ← allocateVma(state.task, 16 ∗ 2^20)
36:     stackTop ← state.stack + 16 ∗ 2^20
37:     outputSpace ← stackTop − len(task.output)
38:     inputSpace ← outputSpace − len(task.input)
39:     inputSpace[0 : len(task.input)] ← task.input[0 : len(task.input)]
40:     state.proc.regs.rbp ← inputSpace
41:     state.proc.regs.rsp ← inputSpace
```

```
42:     state.proc.regs.rip ← task.fun
43:     state.proc.regs.rdi ← inputSpace
44:     state.proc.regs.rsi ← outputSpace
45:     scheduleTask(state.proc)

46: procedure protectMemory(proc)                    ▷ setup virtual memory
47:     for each vma in proc.vmas do
48:         if vma.flags & VM_WRITE then
49:             clearPages(proc, vma)
50:             vma.page_fault_handler ← pageFault
51:             vma.vm_private_data ← proc

52: procedure pageFault(addr)                        ▷ page fault handler
53:     state ← findVma(current, addr).vm_private_data
54:     if state.touched_pages is empty then
55:         state.start_version ← state.ctx.version
56:     page ← pageTableWalk(state.ctx.parent_task, addr)
57:     if page exists then
58:         add addr to state.touched_pages
59:         return page
60:     else
61:         return NO_PAGE

62: procedure validate(ctx, state)                   ▷ pre-commit validation
63:     if state.start_version = ctx.version then
64:         return true
65:     for each addr in state.touched_pages do
66:         if ctx.page_versions[addr] > state.start_version then
67:             return false
68:     return true

69: procedure commit(ctx, state)                     ▷ commit speculative state
70:     for each addr in state.touched_pages do
71:         newP ← pageTableWalk(state.task, addr)
72:         oldP ← pageTableWalk(ctx.parent_task, addr)
73:         if newP ≠ oldP then
74:             pageTableUpdate(ctx.parent_task, addr, newP)
75:             ctx.page_versions[addr] ← ctx.version + 1
76:     ctx.version ← ctx.version + 1
77:     outputSpace ← state.stack + 16 * 2^20 − len(task.output)
78:     task.output[0 : len(task.output)] ← outputSpace[0 : len(task.output)]
79:     flushTlb(ctx.parent_task)
```

6.1.2 User-Space Interface

The interface for starting speculative parallel execution in K-TLS is identical to the one of U-TLS (cp. Section 5.1). The difference in the implementation is that the task list is not processed, but simply passed on to the kernel module via an *ioctl* call. This call returns the number of tasks which were executed and successfully committed in the kernel. If this number is smaller than the number of tasks in the list, the remaining tasks are executed in user space sequentially (lines 4 to 6).

6.1.3 Kernel-Space Interface

The kernel-space routine that implements the *ioctl* call is KTLSRUN, which receives the task list from user space. It then allocates a *KTLSContext* to store information about the execution of the current task list, and an array of N *TaskState* structures, one for each task.

Then, the individual tasks are forked as described in the next section. Afterwards, they are committed in order (lines 18 to 27). Just as for U-TLS, the parent first waits for the completion of the task. Then, it validates the recorded changes of the task (see Section 6.1.5). If this validation fails, the task is re-spawned with an up-to-date view of all memory changes committed so far (lines 22 to 23). This re-spawned process does not need to be validated, since no other task committed since its start. It is checked however, that the last spawned process for this task (original or re-spawned) did complete the execution of the user code (line 24). If this check fails, it means that the process either received a signal because of illegal memory accesses, or exited

explicitly by calling `abort` or triggering an assertion. If all checks succeed, then the task's memory changes are committed to the main process (see Section 6.1.5).

6.1.4 Execution of Speculative Tasks

After a new child process is forked from the main process (line 32), its memory is made inaccessible by iterating over all writable *virtual memory areas (VMAs)* and clearing all pages corresponding to those VMAs from the page table. Also, our own page fault handler is registered for those VMAs. Then, a hash map for all accessed memory pages is allocated, and a new memory area for the stack is created with a size of 16 MB. This memory region is unprotected, and pages are allocated on demand. The top of the stack is initialized with a copy of the input data of the task, such that accessing this data does not trigger a page fault. Also, space for the output of the task is reserved there. The content of this space will be copied to the original output location of the task during the commit phase. Then, the registers of the newly forked task are set such that the process—once scheduled—will use the newly allocated stack for its stack frames (remember that the stack grows downwards), and will execute the user code with the input and output spaces on the stack as arguments (lines 40 to 44). Finally, the task is scheduled for execution by an idle core.

During execution of a speculative task, the kernel module only becomes active if page faults or system calls happen (see Section 6.1.6). Page faults are handled by a general routine in the kernel, which looks up the VMA of the faulting address, and calls the page fault

handler registered for this VMA. For the VMAs protected by K-TLS, this will be the pageFault routine (line 52). It first resolves the memory address to the VMA of the current process, to get a pointer to the *TaskState* structure of the current task. If it then finds that this is the first page fault in this task, it sets the *start_version* of the task to the committed version in main memory (*task.ctx.version*), which is identical to the index in the task list of the last committed task plus one. This ensures that there are no false conflicts reported with tasks that commit between the fork point and the first memory access of the current task. We could further reduce the number of false conflicts by storing the *read version* for each single accessed page. It is unclear, however, whether the additional resources for this would pay off. We thus leave this for future work. The page fault handler proceeds by looking up the page in the page table of the parent task (line 56). If the page exists, its address is added to the set of touched pages, and it is returned as the outcome of the page fault. The operating system then adds the page to the page table of the current task, or creates a private copy of it in the case of a write page fault. If no corresponding page is found in the parent process, *NO_PAGE* is returned by the page fault handler, which results in a segmentation fault being triggered.

6.1.5 Validation and Commit

The pre-commit validation of a task can be cut short if no task committed since the first memory access of the task (line 63). Otherwise, all pages which have been touched by the process (read or written) are checked against the *page_versions* map in the context to detect if

any of them was modified since the start of the task. If no conflict is found during this validation, then the actual commit phase can start.

During commit, for each page which was accessed by the child task, the kernel module compares the physical pages this address maps to in the task and the parent process (lines 71 to 72). If those pages differ, we can conclude that the kernel created a private copy of the page via the *copy-on-write* semantics of shared pages, thus we know that the page was modified. In this case, we update the page table of the parent process to map *addr* to the modified page *new*, and register the new version of this page in the *ctx.page_versions* map. Note that the physical page in the parent cannot have changed during execution of this task by committing other tasks, because otherwise a conflict would have been reported. After committing all changed memory pages, we update the overall memory version (line 76) and copy the direct output of the task (cf. Section 5.1) from the child's stack back to the parent (lines 77 to 78). Also, the *translation lookaside buffer (TLB)* of the parent process is flushed such that it is refilled by the hardware with the modified page table entries.

6.1.6 Handling of System Calls

TLS systems often promise full isolation of speculative tasks, but this merely includes memory effects. As a kernel module, K-TLS also provides full isolation in the presence of system calls like I/O or low-level memory operations like `mmap` or `mprotect`. To this end, the kernel module manipulates the system call table which stores the pointers to the kernel-mode system call handlers. All entries

corresponding to forbidden system calls[1] are rewritten such that a K-TLS routine is invoked on an attempt to perform a system call. This routine first checks whether the current process executes a speculative K-TLS task. If not, the routine jumps to the original system call handler. This check only requires a small number of memory accesses, and produces no observable overhead. If the process executes a speculative task, the task is immediately aborted. Aborted tasks are later detected as invalid executions, hence they will re-execute sequentially.

6.1.7 Optimizations

For clarity of presentation, we slightly simplified some of the implementation details in the previous sections. For performance reasons, the actual implementation sometimes deviates from the description in the text. We give an overview over these optimizations below.

In the code shown in Algorithm 6.1, the main process first forks each task, and then sets up the forked task for speculative execution. The real implementation actually does most of the setup in the forked task itself, thereby executing it in parallel to the setup of other tasks and removing its delay from the critical path. This is achieved by setting the instruction pointer initially to the newly allocated stack area, and having the stack page fault handler execute the setup on the first page fault (this handler otherwise just returns a newly allocated page). The parent then only copies the current process, sets up the

[1] A small number of system calls is white-listed because they do not modify any state in their process, e.g. `nanosleep` or `gettimeofday`.

stack VMA and the instruction pointer, and schedules the task for execution.

We also optimize the actual cloning of the process: Instead of performing a deep copy of the parent page table—as it is usually done in a fork—and then clearing all page table entries which belong to protected memory (line 49), we just skip copying the respective VMAs and associated page table entries, and allocate new VMAs during the aforementioned setup. The new VMAs refer directly to the custom page fault handler (line 52) which will be called by the kernel if a page fault happens inside one of these VMAs. Similarly, the *file descriptor table* does not need to be copied, since system calls working on these open files are prohibited anyway.

Since spawning new processes still requires significant time (see Section 5.3.1.1), we avoid repeated forking by reusing finished processes for the execution of later tasks. To this end, after committing or rolling back a finished task, the corresponding process does not exit. Instead, it clears all page table entries belonging to writable VMAs, puts itself in a waiting queue and sleeps until it is woken up to either execute another speculative task, or because the parent process is exiting. After waking up, it checks that its VMAs are still in line with those in the parent process, and updates them otherwise.

6.2 Improving Granularity via Instrumentation

As discussed in Section 2.2.5, a general problem of isolating speculative from non-speculative memory by protecting and communicating whole

memory pages is the coarse granularity implied by these approaches. Since only the first access to each page is observed, the validation and commit phases have no information about which memory regions inside the respective page have been read or written. By keeping the version of each memory page at the starting time of a transaction, the granularity for memory writes can be improved by *diffing* the speculative against the original memory page. This however only solves the problem for write-after-write conflicts on pages *which were not read* by any of the participating transactions. As we also allow read accesses on any write-enabled page, this approach would not work in our setting.

In order to fully solve the problem of granularity, we propose to add *code instrumentation* to the K-TLS approach, in order to track memory accesses at runtime. We thus create a system with tunable granularity, but performance comparable to the fastest known systems. The design and implementation of this addition—which we call K-TLS^{+}—is described in the remainder of this section. This work was conducted together with Daniel Birtel and is partially described in his Bachelor's thesis [7].

6.2.1 Overall Design

Previous work (see Section 2.2.5) has shown that the *ideal granularity* differs greatly between applications, as it heavily depends on the memory access patterns of the transactional tasks. Also, the granularity should not be fixed within one program run as it varies also within one application. Therefore, we keep the choice of granularity dynamic.

As we want to be able to switch seamlessly between the TLS implementation, K-TLS$^+$ uses the same interface as K-TLS and U-TLS (see Section 5.1). The task list is extended by an integer field which specifies the granularity used for instrumenting the code. Since typically a compiler is used for code instrumentation, it can easily fill in the chosen granularity when generating the code for spawning the speculative tasks.

The idea of K-TLS$^+$ is simple: We associate two bits for each *block* of memory as defined by the granularity. One bit stores whether the corresponding block was read by the transaction, the other one whether it was modified. These bits are then used during validation, such that we now check for memory conflicts *per memory block* instead of *per memory page*, thereby reducing the granularity of conflict checking.

In order to keep the overhead of the instrumented code low, we restrict ourselves to granularities that are *powers of two*. The system supports granularities between 2^0 and 2^{12}, where the latter merely exists for comparison purposes, as it does not improve granularity over K-TLS (2^{12} is the size of a memory page on the x86 architecture).

The access bits are stored in *shadow memory* which is automatically allocated by the kernel module. This shadow memory is located at a fixed offset (see Section 6.2.2), and all accessible memory is mapped linearly into this area, such that the computation of the shadow memory address based on the accessed memory address is very cheap (see Section 6.2.3). Note that we make use of *lazy page allocation* for the shadow memory, such that we only allocate the shadow memory pages which are actually accessed within each transaction.

6.2.2 Accessing the Shadow Memory

The concept of *shadow memory* [68, 69] is used by different tools to store additional information for every byte or larger block of memory. This is particularly interesting for automatic approaches which should not or can not alter the layout of the memory objects themselves. Different schemes have been proposed to map arbitrary objects in memory to corresponding shadow memory. Traditional approaches like Valgrind [70, 71] or Dr. Memory [11] use single- or multi-level *translation schemes* similar to hash tables or multi-level page tables to implement the mapping. They speed up the lookup procedure by optimizing for the case that the respective entry in the lookup structure already exists, and accept larger overhead for the rare case that new data needs to be allocated. This is achieved by initializing all pointers such that an attempt to dereference them leads to a signal, which is then caught by a special signal handler which sets up the respective data. This allows to skip all explicit checks for non-existing data. Using this technique, Valgrind and Dr. Memory achieve small single-digit overhead factors.

If the address space is large enough though—which is the case nowadays on 64 bit architectures—the overhead can be further reduced by completely eliminating memory loads to determine the address of the shadow memory. In order to achieve this, the shadow memory region must be located at a fixed offset, and be big enough to hold the metadata for all application memory. In this case, the shadow memory is just a projection of the real memory by shifting and scaling the memory address. Hence, computing the shadow memory address requires just a small number of arithmetic instructions. Such

an approach is implemented for example in the *AddressSanitizer* tool [105].

In the case of K-TLS$^+$, the shadow memory can actually be much smaller than the corresponding application memory, since we only need to store two bits of information for each *memory block* as defined by the granularity. Hence each byte of shadow memory stores the information for four blocks of memory. Even though the x86-64 architecture specifies memory addresses to be of 64 bit, current hardware only supports 48-bit physical addresses [1, 65]. The Linux kernel in its current version further restricts this to 47 bit which are addressable from user space [56]. Hence, the size of the shadow memory area required to cover all address is

$$bytes_{shadowMemory} = \frac{2^{47}}{4 * bytes_{memoryBlock}}$$

For the finest granularity ($bytes_{memoryBlock} = 1$), the shadow memory area has to hold 2^{45} bytes. For K-TLS$^+$, we chose the address `0x100000000000` ($= 2^{44}$) for the start of the shadow memory. This area is typically empty, as it is located 17 terabytes ahead of the starting address of the heap, which grows upwards.

6.2.3 Code Instrumentation

K-TLS$^+$ relies on the user-space software to update the shadow memory correctly during execution of speculative code. Similar to STM systems, this can either be done manually by only accessing potentially conflicting memory locations via special library functions,

or automatically by an instrumenting compiler. Since the main target of this work is automatic parallelization, we only consider the automatic approach in this work.

The instrumentation is performed by iterating over a copy of the transactional code and augmenting all memory operations by code to set the corresponding bit in shadow memory. Accesses to the local stack are skipped since they can never conflict with other transactions. When function calls are encountered in the speculative code, and the callee is statically known, a copy of the respective function is instrumented recursively, while instrumented functions are reused for all other call sites to the same function. If indirect function calls are encountered and the *Sambamba* runtime system is available, then the call is replaced by a callback to the runtime system to resolve the respective function and dynamically create the instrumented copy of the function if it is not available yet. A small callsite-local cache is used to store the mapping from un-instrumented to instrumented functions. This cache is updated after each callback to the runtime system (see Section 4.2 for details).

Some intrinsic functions like `memset` and `memcpy` are handled separately by marking the respective shadow memory bits explicitly (see Section 6.2.3.2). The execution of any other intrinsic call, which is not marked as `readnone` in LLVM (specifying that the respective operation does not read or modify any memory) and is not explicitly whitelisted leads to a rollback of the respective task. The same applies to any external function call, like for example to the C library.

The remainder of this section details the instrumentation performed for different LLVM instructions.

6.2.3.1 Load and Store Operations

The most common instruction that needs to be instrumented is a `load` or `store` instruction. In LLVM, each such instruction takes the address of the memory object, and statically knows the type of the accessed element. From the type, we can directly derive the size of the element in terms of bytes. Also, each load or store carries information about the guaranteed alignment of the memory object.

For most cases, it is statically known that only one *memory block* is accessed, so only a single *meta data unit (MDU)* needs to be updated. A single stream of instructions can be emitted to compute the address of the shadow memory byte and the index of bit to be updated within this byte, and then load the respective byte, set the computed bit, and store it back. In this case, no control flow is involved, and a single load and store is sufficient to update the information in shadow memory.

This single-MDU update is sufficient if (a) the size of the accessed element is not greater than a memory block, and (b) the alignment is at least as large as the size of the element. If any of these two conditions is not fulfilled, more complex code has to be emitted to potentially update more than just one bit. Measurements on our benchmarks have shown that for granularities of word-size or larger most instrumented locations just require a single-MDU update (see Table 6.3 on page 156). For the remaining cases the code is more complex, as potentially more that one word in the shadow memory has to be updated. This requires adding control flow. For this code, we fall back to the more general case of updating a dynamically sized memory region, as described in Section 6.2.3.2. As the size

Algorithm 6.2 Computations done for a single-bit shadow memory update. See Algorithm 6.3 for the actual LLVM instructions emitted to implement this computation.

$$
\begin{aligned}
metadataUnit &= \left\lfloor \frac{address}{bytes_{memoryBlock}} \right\rfloor \\
shadowByteAddr &= \texttt{0x100000000000} + \left\lfloor \frac{metadataUnit}{4} \right\rfloor \\
bitInShadowByte &= 2 * (metadataUnit \texttt{ mod } 4) + (isStore \ ?\ 1 : 0) \\
oldShadowByte &= load(shadowByteAddr) \\
newShadowByte &= oldShadowByte \mid 2^{bitInShadowByte} \\
& store(shadowByteAddr,\ newShadowByte)
\end{aligned}
$$

of the memory to be updated is always known for loads and stores, we always take advantage of the optimizations described there, and hence never generate the full code necessary for the general case.

As described in Section 6.2.2, the shadow memory is located in such a way that computing the shadow memory address for any given memory address does not require additional memory operations. Instead, it is computed via a number of arithmetic operations as described in Algorithm 6.2.

First, the index of the *metadataUnit* is computed by dividing the actual memory address by the size of a *memory block* (which is always a power of two). Since each metadata unit consists of two bits, four of these units are stored in each byte of shadow memory, as computed in *shadowByteAddr*. The bit which represents the corresponding memory operation inside of this shadow memory byte is computed as the index of the metadata unit modulo 4 if it is a reading memory operation, and one more for a writing memory operation. Then,

Algorithm 6.3 Instrumentation of one store instruction in LLVM. The `%granularity` value is a fixed constant during instrumentation, so some of the computations are folded by later optimizations.

```
store <type> %value, <type>* %address

  ↓

store <type> %value, <type>* %address
%addressInt = ptrtoint %address to i64
%metadataUnit = lshr i64 %addressInt, %granularity
%metadataUnitInByte = and i64 3, %metadataUnit
%readBitOffsetInByte = shl i64 %metadataUnitInByte, 1
%writeBitOffsetInByte = add i64 %readBitOffsetInByte, 1
%updateMask = shl i8 1, %writeBitOffsetInByte
%metadataUnitOffset = lshr i64 %metadataUnit, 2
%shadowByte = add i64 0x100000000000, %metadataUnitOffset
%shadowByteAddr = inttoptr i64 %shadowByte to i8*
%oldShadowByte = load i8* %shadowByteAddr
%newShadowByte = or i8 %oldShadowByte, %updateMask
store i8 %newShadowByte, i8* %shadowByteAddr
```

the shadow memory byte is loaded, and the respective bit is set by conjunctively combining the loaded byte with a mask where only that one bit is set. This new value is then written back to memory.

The actual computation as inserted via instrumentation of the LLVM code is given in Algorithm 6.3. Note that the `ptrtoint` and `inttoptr` just convert between different types on the LLVM level, and hence are no-ops in the generated machine code. Also, `%granularity` is a constant value at the time of instrumentation, so later optimization phases often further optimize the instruction sequence by combining redundant instructions. Statically known alignment information might further reduce the amount of computations.

If the memory address is fully known at link time—as it is the case for global variables—the arithmetic operations for computing both the address of the shadow memory byte as well as the mask for the

updated value are folded into so called *constant expressions*, which are translated into constants at link time.

6.2.3.2 Updating Larger Memory Blocks

The above mentioned code sequences only work for those cases where we can statically prove that only a single memory block will be touched by the memory access—and hence, only a single bit needs to be set in shadow memory. If the accessed memory region is too large, or not aligned properly, or if its size is not statically known—like for example on `memset` calls—, more general code is emitted which is able to efficiently update an arbitrary number of MDUs.

Algorithm 6.4 shows the pseudo-code implementation of such a shadow memory update involving potentially more than one MDU. Since it contains several branches and a loop with unknown trip count, it is much more heavy-weight than the simple procedure for updating a single MDU as shown in Section 6.2.3.1. In the first step, the index of the first and the last involved MDU is computed. Based on this, we compute the address of the first and the last shadow memory word to update, and the mask to update just the upper or lower part of these words as appropriate. Now in the special case that only a single word needs to be updated, we combine both computed masks and update this one word. Otherwise, we update the first and the last word, and then continue updating all full words in between. The check for $\mathit{firstShadowWord} < \mathit{lastShadowWord}$ is needed for the special case that the length of the memory block is 0, which can happen for intrinsic calls like *memset*.

Algorithm 6.4 Pseudo-code implementation of a multi-MDU update. See Algorithm 6.5 for the actual LLVM instructions emitted to implement it.

```
1: input: void* startAddress, void* endAddress, bool isStore
2: global: uint64_t *shadowMemory

3: bitMask ← 0x5555555555555555 ≪ (isStore ? 1 : 0)
4: firstMDU ← (uint64_t)startAddress ≫ granularity
5: lastMDU ← ((uint64_t)endAddress − 1) ≫ granularity
6: firstShadowWord ← shadowMemory + (firstMDU ≫ 5)
7: lastShadowWord ← shadowMemory + (lastMDU ≫ 5)
8: firstWordMask ← bitMask ≪ (2 ∗ (firstMDU & 31))
9: lastWordMask ← bitMask ≫ (62 − 2 ∗ (lastMDU & 31))
10: if firstShadowWord == lastShadowWord then
11:     updateMask ← firstWordMask & lastWordMask
12:     ∗firstShadowWord |= updateMask
13: else if firstShadowWord < lastShadowWord then
14:     ∗firstShadowWord |= firstWordMask
15:     ∗lastShadowWord |= lastWordMask
16:     for word ← firstShadowWord + 1 to lastShadowWord − 1 do
17:         ∗word |= bitMask
```

The actual bitcode emitted, as shown in Algorithm 6.5, contains six new basic blocks which are inserted between the instrumented instruction and its successor. It closely implements the code shown in Algorithm 6.4, with the check for an empty length moved further up to avoid partial dead code. Depending on static information computed at instrumentation time some of the branches can be skipped completely. In the case of an unaligned or too large load or store, where the size of the accessed memory is always known, the check for zero length and the branch depending on it can be skipped. Also, for all accesses up to $32 * bytes_{memoryBlock} + 1$, we know that no more than two words of shadow memory will be updated, since not more than 33 MDUs

Algorithm 6.5 LLVM instructions emitted to implement a multi-MDU update. Depending on further static information, the code can be minimized by later optimization passes.

```
  call void @llvm.memset(i8* %address, i8 %val, i64 %len)
    ↓
  call void @llvm.memset(i8* %address, i8 %val, i64 %len)
  %isZeroLen = icmp eq i64 %len, 0
  br i1 %isZeroLen, label %continuation, label %updateMDUs

updateMDUs:
  %startAddr = ptrtoint %address to i64
  %endAddr = add i64 %startAddr, %len
  %firstMDU = lshr i64 %startAddr, %granularity
  %endAddrMinusOne = sub i64 %endAddr, 1
  %lastMDU = lshr i64 %endAddrMinusOne, %granularity
  %firstWordOffset = lshr i64 %firstMDU, 5
  %firstWordAddr = getelementptr i64,
    i64* (inttoptr i64 0x100000000000 to i64*), i64 %firstWordOffset
  %lastWordOffset = lshr i64 %lastMDU, 5
  %lastWordAddr = getelementptr i64,
    i64* (inttoptr i64 0x100000000000 to i64*), i64 %lastWordOffset
  %firstMDUindex = and i64 %firstMDU, 31
  %firstMDUbitnr = shl %firstMDUindex, 1
  %firstWordMask = shl 0xaaaaaaaaaaaaaaaa, %firstMDUbitnr
  %lastMDUindex = and i64 %lastMDU, 31
  %lastMDUbitnr = shl %lastMDUindex, 1
  %lastWordNonAffectedBits = sub i64 62, %lastMDUbitnr
  %lastWordMask = lshr 0xaaaaaaaaaaaaaaaa, %lastWordNonAffectedBits
  %isSingleWordUpdate = icmp eq i64* %firstWord, %lastWord
  br i1 %isSingleWordUpdate,
    label %updateSingleWord, label %updateMultipleWords

updateSingleWord:
  %updateMask = and i64 %firstWordMask, %lastWordMask
  %oldSingleWord = load i64* %firstWordAddr
  %newSingleWord = or i8 %oldWord, %updateMask
  store i64 %newSingleWord, i64* %firstWordAddr
  br label %continuation

updateMultipleWords:
  %oldFirstWord = load i64* %firstWordAddr
  %newFirstWord = or i8 %oldFirstWord, %firstWordMask
  store i64 %newFirstWord, i64* %firstWordAddr
  %oldLastWord = load i64* %lastWordAddr
  %newLastWord = or i8 %oldLastWord, %lastWordMask
  store i64 %newLastWord, i64* %lastWordAddr
  br label %updateWordsInBetween

updateWordsInBetween:
  %prevWord = phi i64* [%firstWord, %updateMultipleWords],
                       [%nextWord, %updateWordsInBetween]
  %nextWord = getelementptr i64, i64* %prevWord, i32 1
  %finished = icmp eq %nextWord, %lastWordAddr
  br i1 %finished, label %continuation, label %updateNextWord

updateNextWord:
  %oldWord = load i64* %nextWord
  %newWord = or i8 %oldWord, 0xaaaaaaaaaaaaaaaa
  store i64 %newWord, i64* %nextWord
  br label %updateWordsInBetween

continuation:
  [...]
```

are involved. We can thus skip the whole loop consisting of the blocks *updateWordsInBetween* and *updateNextWord*. Those optimizations could potentially also be performed by later optimization stages, but we implement them directly in the instrumentation pass to avoid unnecessary overhead, and avoid relying on sophisticated compiler analyses (at least for the second case).

6.2.4 Changes to the Kernel Module

Apart from the instrumentation of the target code in order to explicitly mark accessed memory regions, also changes in the kernel module are needed. For each task, a special *virtual memory area (VMA)* for the shadow memory is allocated and it is used to improve the granularity during validation and commit of a task's changes. This section describes the changes to the kernel module in detail.

6.2.4.1 Task Setup

During the setup of a task, before any of the actual code is executed, a VMA for the shadow memory is created. Its size is determined by the granularity according to the formula given in Section 6.2.2:

$$shadow\_memory\_size = 1 \ll (45 - granularity)$$

This accounts for the addressable address space of 2^{47} bytes, and the ability to store four MDUs in one byte. The VMA is allocated directly after the setup of the task's virtual memory descriptor, hence in the child process itself (see Section 6.1.7). If a task is reused, we ensure that the VMA for the shadow memory is *at least* as large

as required for executing this task, i.e. we increase it if necessary, but never decrease. Note that for this VMA no actual memory is allocated yet, just the kernel structure is created to allow the user code to access this memory. The memory pages are allocated on demand during execution, and are initialized to zero.

6.2.4.2 Validation

During validation, the information stored in the shadow memory is used to prove tasks memory-conflict-free, even though they accessed the same memory page. For each page, we first run the usual validation of K-TLS (see Section 6.1.5). Only if this validation detects a conflict with a previous task, we inspect the shadow memory of both tasks for the respective page. We thus keep the shadow memory of committed tasks present until all tasks which overlapped in execution did also commit.

The validation against the shadow memory of another task is performed by iterating over the shadow memory word by word and checking whether any MDU was written in the previous task and read or written by the current task. This comparison can be implemented using bit operations:

```
if (current_word & (other_word | (other_word >> 1)) != 0) {
  return MEMORY_CONFLICT;
}
```

If the granularity is large enough such that one shadow word covers more than one memory page, then this simple check could return true even though there is no conflict on the page we are currently checking, but on another (neighboring) one. However, since we are

only interested in the outcome for all modified pages, it is ok to return *MEMORY_CONFLICT* also in this case.

6.2.4.3 Commit

In K-TLS$^+$, the commit procedure works very different from the one of K-TLS. Instead of moving physical pages between the child process and the parent by copying page table entries, we actually copy the changed memory content from the child's physical pages to the parent's. Even though for some of the changed pages it might also be possible to use K-TLS' approach, we decided against this in order to be able to compare both approaches against each other. In some cases copying might even be faster, especially if only a small portion of the page was actually modified. Also, the complexity of the implementation is simplified a lot, and we do not need to flush the *translation lookaside buffer (TLB)* of the parent process after commit, since no page table entries are changed.

For the commit itself, we iterate over the shadow memory of all changed pages word by word and find *ranges* of modified memory within that page. This procedure is again implemented using bit operations and the *count trailing zeros* built-in, for which a hardware instruction exists on most architectures. When a range is finished, we use *memcpy* to copy over the respective part of the child's physical memory page to the parent's. As the evaluation of this approach showed surprising peaks in the commit time for low granularities on some benchmarks (up to three times larger than K-TLS' commit time), we investigated this and found out that it happens if many small regions are found within one page, and we hence call *memcpy*

very often. It turned out that the performance can be improved by more than a factor of two by replacing the *memcpy* by a macro which first checks whether the length of the region is exactly 1, 2, 4 or 8 bytes and using a simple memory load/store combination for those cases. As this check generally does not hurt the performance significantly, we keep this implementation.

6.3 Evaluation

Since K-TLS requires loading a kernel module into the operating system of the evaluation system, all evaluation is performed in a virtual machine. Intel virtualization extensions (VT-x) are enabled to minimize the runtime impact of virtualization.

We verified the measurements in the VM against executions directly on the host system, and checked that the time measures match. The host system is the same as before. It is equipped with a quad-core Intel i7 870 CPU running at 2.93 GHz and 16 GB of main memory, and is running the Linux kernel in version 4.1.12. The virtual machine has access to all four CPUs, and 8 GB of memory. For each benchmark, we run at least 10 runs and report the arithmetic mean over the first three quartiles. No other processes were executing on both the host and the guest system.

We compare the K-TLS system against U-TLS as described in Chapter 5, and include TinySTM$^+$ (see Chapter 4) for reference. In Section 6.3.3 we compare K-TLS$^+$ against K-TLS and also U-TLS.

6.3.1 TLS Overhead

In this section we measure the performance of the primitive operations of K-TLS and compare it against the same operations in U-TLS. Analogous to Section 5.3.1, we measure the time for spawning tasks, the overhead that the speculation system induces during the parallel execution of the tasks, and the validation and commit time. The microbenchmarks are the same as introduced in Section 5.3.1.

6.3.1.1 Spawning Tasks

Similarly to U-TLS, K-TLS spawns individual processes to execute the individual tasks. By implementing the forking and all setup of the processes directly in the kernel, it not only saves a lot of context switches (for example to protect each VMA in the process), but it also executes less redundant work. Instead of creating a deep copy of the whole page table and then clearing it again via *mprotect* system calls, K-TLS already skips the respective VMAs when cloning the page table. Instead, it installs empty VMAs afterwards and registers as the page fault handler for those areas.

We execute the benchmark already shown in Section 5.3.1.1 with U-TLS, in order to evaluate the effect of this optimization and the other minor changes mentioned in Section 6.1.7. The results are shown in Figure 6.1. K-TLS takes about 2 ms for spawning the first task, and between 0.5 and 0.6 ms per additional task. Since the page table entries for all writable regions are omitted during the fork of the child process, the spawn times in K-TLS are independent of the amount of memory allocated in the process. We therefore show just

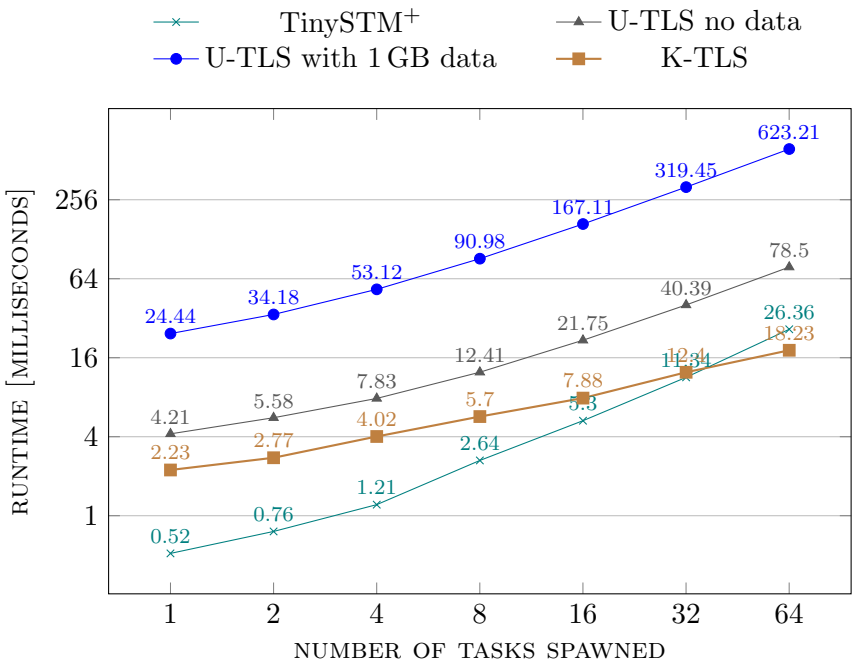


FIGURE 6.1: Overhead comparison for spawning a task list of varying size in K-TLS versus U-TLS and multi-threaded systems. K-TLS forks the process directly in the kernel, requiring less context switches than U-TLS for protecting memory. K-TLS takes around 2.2 ms initially plus 0.5 ms per spawned task. Additionally, it reuses tasks once they finish execution, leading to less actually forked tasks for larger numbers.

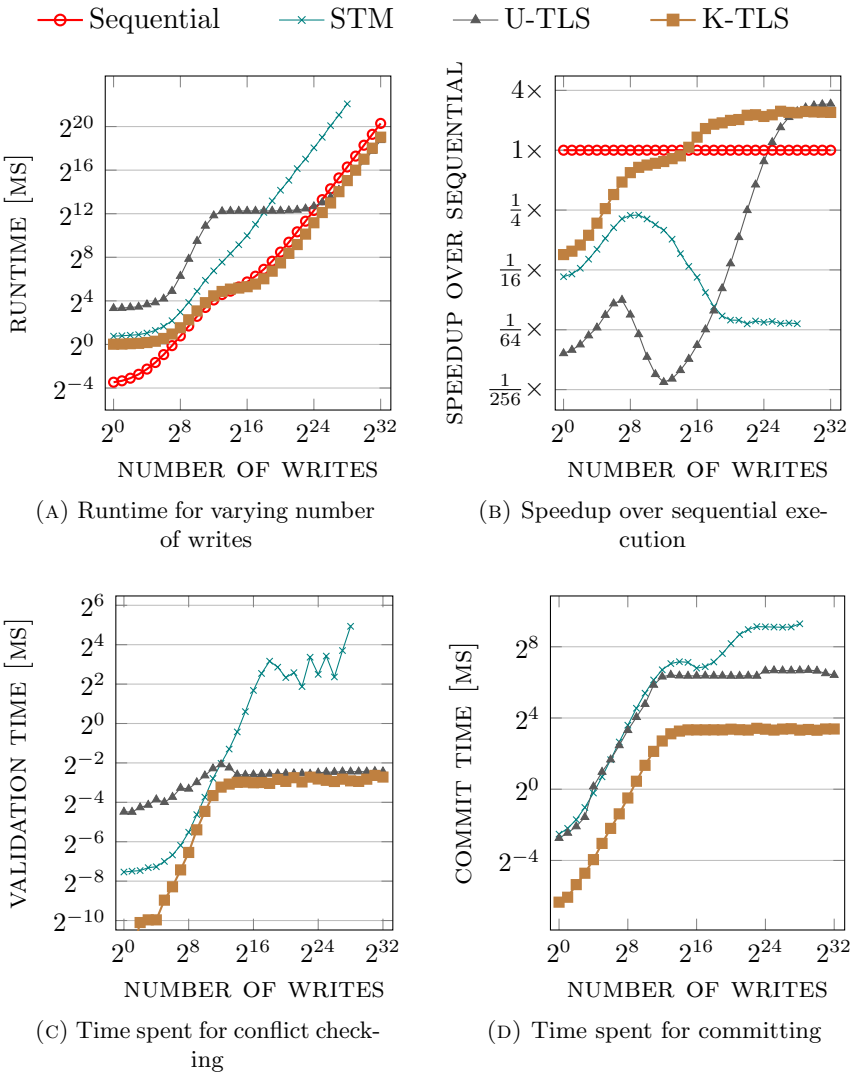
one plot for K-TLS, just as for TinySTM$^+$. The overhead of forking a process in K-TLS is about a factor of two lower than for U-TLS. If the task has access to 1 GB of user data, the factor increases to 11×. Additionally, by reusing the processes after finishing the execution of one task, K-TLS further reduces the cost per additional task to below 0.2 ms, which becomes visible if a larger number of tasks is spawned. For spawning 64 tasks, K-TLS even beats the time it takes

an STM-based system to create the respective threads; K-TLS takes on average 0.28 ms per task while U-TLS takes 1.23 ms to 9.74 ms and TinySTM$^+$ 0.41 ms.

6.3.1.2 Execution Overhead

The next benchmark evaluates the execution overhead of K-TLS. Here, the overhead is mainly caused by handling the page faults which are triggered by the MMU whenever the process executing a speculative task accesses a memory page for the first time. In contrast to U-TLS, the page fault will not result in a segmentation fault which is then handled in user space, but instead, the page fault will directly be handled in kernel space by the K-TLS kernel module. So instead of six to eight context switches (see Section 5.3.1.2), just two are needed: to kernel space and back.

The effect of this improvement can be seen in Figures 6.2a and 6.2b. For a small number of write operations, no parallelization scheme will be able to provide speedup, since the overall sequential execution time is below one millisecond or just slightly above it. However, even in this range, K-TLS performs significantly better than U-TLS. In the mid-range, where U-TLS faces the problem of fragmenting the virtual address space, K-TLS shows consistent performance. This is because no kernel structures are changed when handling the page faults. The new page is directly inserted into the page table, which has null entries for all the non-accessible pages. From 2^{15} write accesses on, K-TLS performs better than sequential execution, even though at this point sequential execution only takes 38 ms overall.



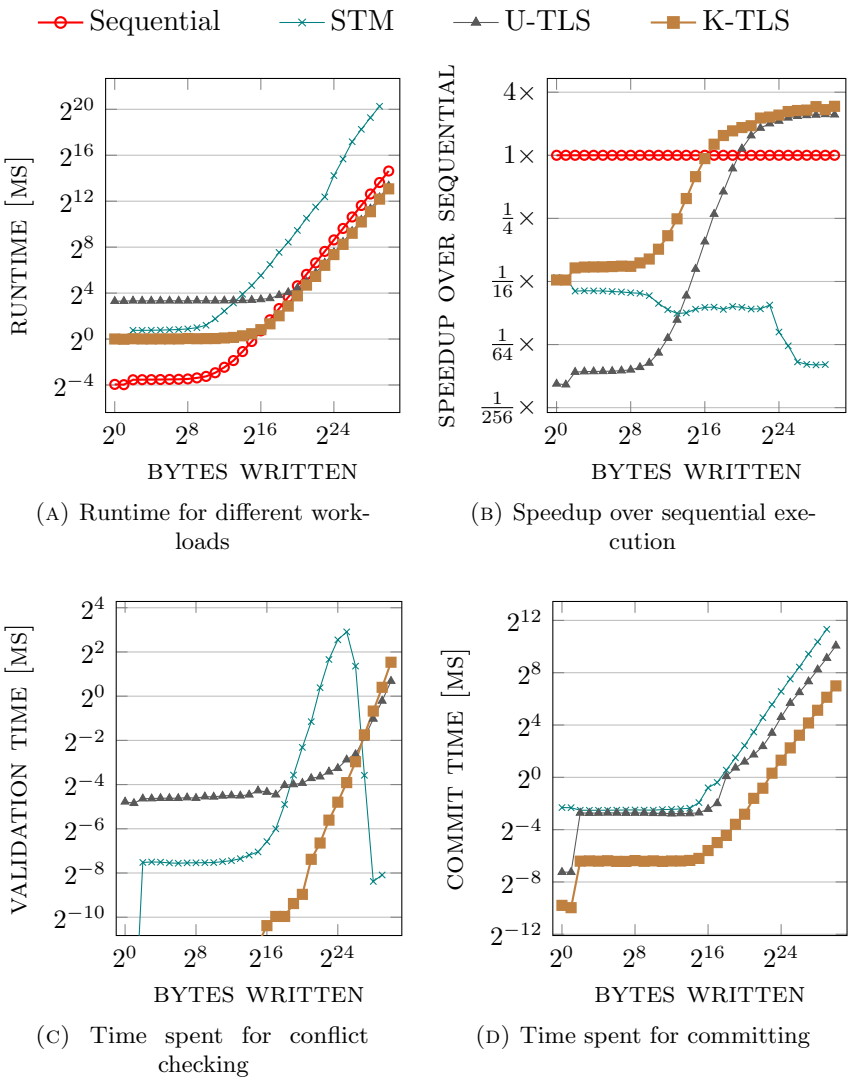
(A) Runtime for varying number of writes

(B) Speedup over sequential execution

(C) Time spent for conflict checking

(D) Time spent for committing

FIGURE 6.2: Performance of the different runtime systems in an artificial benchmark, in which each task randomly updates memory cells within a 16 MB memory block (cf. Figure 5.3). K-TLS performs much better than both U-TLS and TinySTM. It provides speedups over sequential execution for mid-sized to large-sized matrices, and finally comes close to the perfect speedup of 4×.



(A) Runtime for different workloads

(B) Speedup over sequential execution

(C) Time spent for conflict checking

(D) Time spent for committing

FIGURE 6.3: Performance comparison of K-TLS, U-TLS and TinySTM$^+$ in a benchmark in which each task writes linearly to a memory block of varying size (cf. Figure 5.4). Again, K-TLS provides much better performance than U-TLS for small or medium tasks, and is consistently better than TinySTM$^+$. K-TLS commits the written memory block between 5× and 18× faster than U-TLS.

Also for the benchmark which linearly writes a memory block of varying size, shown in Figure 6.3, there is a huge difference between K-TLS and U-TLS performance especially for small to medium-sized tasks. Only if tasks write a really huge memory block, U-TLS is able to catch up. This suggests that for this setting, the TLS overhead is hidden behind the performance loss due to frequent cache misses. However, K-TLS still performs slightly faster, providing a 3.0× speedup for the largest input compared to 2.6× for U-TLS.

6.3.1.3 Validation and Commit

Figures 6.2c and 6.2d show the validation and commit time for the benchmark performing random updates within a constant-size block of memory, Figures 6.3c and 6.3d for linearly writing to memory. Our first observation is that K-TLS performs significantly better than U-TLS for small to medium workloads. Since validation does not require any communication between processes, K-TLS has benefits especially if the amount of data to be communicated is low. As the granularity we use for measuring execution times is one microsecond, we see no time reported at all if the execution takes less than one microsecond. Even though the validation times of U-TLS are also negligible in relation to the overall execution time, it takes more than 50× longer for small to medium workloads. For larger workloads, the validation times of both approaches converge, as they are dominated now by the lookup in the hash table, which has about the same run time in both implementations.

For the commit time, K-TLS also has a huge advantage over the full range of inputs. It varies between 5× (for large tasks) and 11× (for

small to medium tasks) improvement over U-TLS. For linear writes, the commit phase contributes between 1 % and 1.5 % of the overall run time. For random writes, it increases to 12.6 % at 2^{10} writes, and then decreases to less than 0.01 % for huge workloads. This is because the number of written pages increases much faster than for the linear write, but then stabilizes once all 4096 pages have been written.

6.3.2 Usage in Automatic Parallelization

After evaluating the performance in micro-benchmarks to show the benefit of implementing core TLS operations in the kernel, we want to see how this effects the performance of automatically parallelized real-world programs. For this, we run the benchmarks from the *Cilk* suite as introduced in Section 5.3.2. Table 6.1 lists the input parameters used for this evaluation, and some basic characteristics of the execution.

Figure 6.4 shows the speedups of K-TLS and U-TLS over sequential execution of the eight programs. We observe that while U-TLS is only able to speed up five of the programs, K-TLS provides speedups for seven of them. Also, K-TLS is superior to U-TLS in all cases except for spacemul, where they are on a par. The independent two-sample t-test verifies that the difference between K-TLS and U-TLS is significant for each of the other programs ($p \leq 0.01$). The geometric mean of the speedup of K-TLS is 2.02×, while U-TLS shows a slowdown of 2.69×. This sums up to a 5.45× speedup of K-TLS over U-TLS.

TABLE 6.1: Characteristics of eight programs from the *Cilk* program suite, automatically parallelized using the *Sambamba* framework [112]. All programs require speculation, either because of possible side effects like termination in parallel tasks, because of real data dependences, or because of imprecision in the static analyses. The performance numbers are shown in Table 6.2.

program	input size	number of speculative tasks	number of rollbacks	percentage of rollbacks	avg. number of read pages	avg. number of written pages
Cilksort	4194304	8	0	0 %	6148	4096
Fft	4194304	18	1	5.6 %	7769	1294
Heat	4096x1024x100	204	2	1.0 %	8088	4056
Lu	2048	756	147	19.44 %	49	18
Matmul	2048	2	0	0 %	8195	2048
Plu	2048	150	0	0 %	327	135
Spacemul	2048	8	0	0 %	6149	2048
Strassen	2048	8	0	0 %	6149	2048

Interestingly, both U-TLS and K-TLS still provide speedup for the *Lu* program, where the percentage of rollbacks is close to 20 %. These rollbacks are caused by the many small tasks which operate on data which does not span multiple pages, resulting in false conflicts being detected. Note that the average runtime of one task is only 16.9 ms in this benchmark, and the number of read and written pages is the lowest of all programs. This suggests that virtual-memory based TLS systems even provide speedups for low to medium sized parallel tasks. The programs with the lowest performance are *Heat* and *Fft*, which

TABLE 6.2: Performance of eight programs from the *Cilk* program suite automatically parallelized using the *Sambamba* framework [112]. Characteristics of these programs are detailed in Table 6.1. Speedups below 1 (like $\frac{1}{S}\times$) represent slowdowns of factor S. K-TLS provides much better performance than user-space TLS in all eight programs.

program	avg. runtime per task	overall runtime sequential	overall runtime K-TLS	overall runtime U-TLS	speedup K-TLS	speedup U-TLS
Cilksort	347.3 ms	2.05 s	0.87 s	1.67 s	2.36×	1.23×
Fft	102.5 ms	1.21 s	0.92 s	87.31 s	1.32×	$\frac{1}{72.3}\times$
Heat	47.9 ms	3.69 s	4.15 s	1305.82 s	$\frac{1}{1.13}\times$	$\frac{1}{354}\times$
Lu	16.9 ms	13.43 s	6.01 s	7.12 s	2.23×	1.88×
Matmul	25764.2 ms	49.62 s	24.82 s	30.77 s	2.00×	1.61×
Plu	99.9 ms	12.95 s	5.88 s	6.95 s	2.20×	1.86×
Spacemul	2814.3 ms	17.85 s	5.14 s	4.98 s	3.47×	3.59×
Strassen	3467.8 ms	21.89 s	7.32 s	58.69 s	2.99×	$\frac{1}{2.68}\times$
				Geometric mean:	2.02×	$\frac{1}{2.69}\times$

also had the biggest slowdowns in the U-TLS system. Section 5.3.2 already provided some insights why these programs suffer that much from the isolated execution for speculation: Those programs do not access a bigger consecutive memory region, but rather access smaller structures spread over the heap. With the characteristics shown in Table 6.1 we can confirm this hypothesis: *Heat* and *Fft* are the two programs with the lowest ratio of runtime by number of accessed

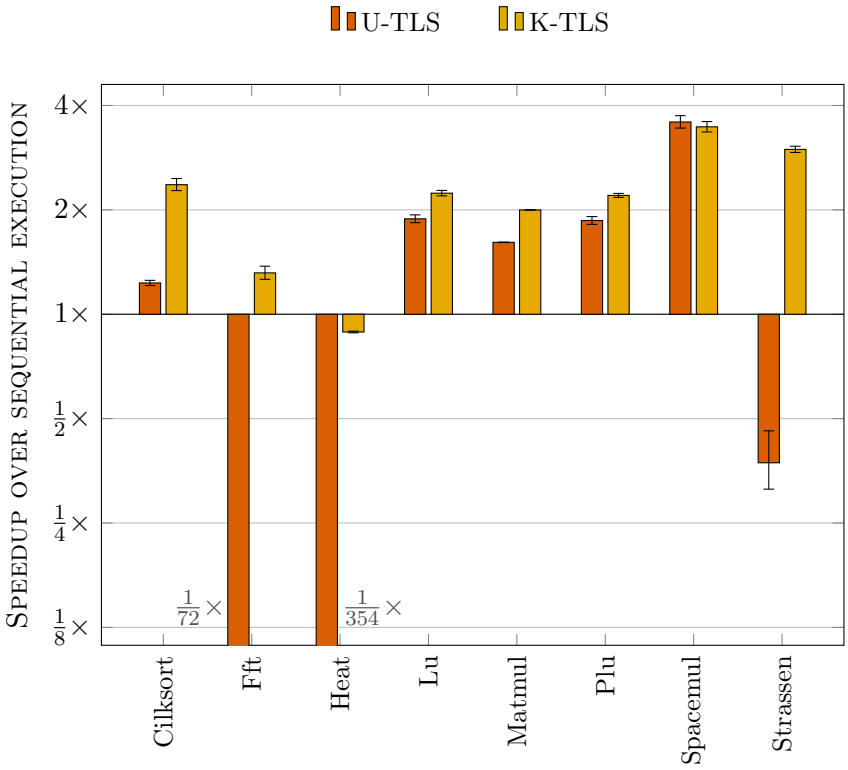


FIGURE 6.4: Comparison of the speedup achieved by automatic parallelization of eight programs from the *Cilk* suite. K-TLS significantly outperforms the state-of-the-art U-TLS in all test cases except for spacemul, where they are on a par. In the geometric mean, K-TLS performs 5.45× better than U-TLS.

pages[2]. This suggests that they often only read small portions of the page, but the TLS system needs to make a private copy of the whole page and in the case of U-TLS also copy back the content of the full page at commit. In K-TLS, we do not need to copy back the content of the page, but there is still a constant overhead per accessed page,

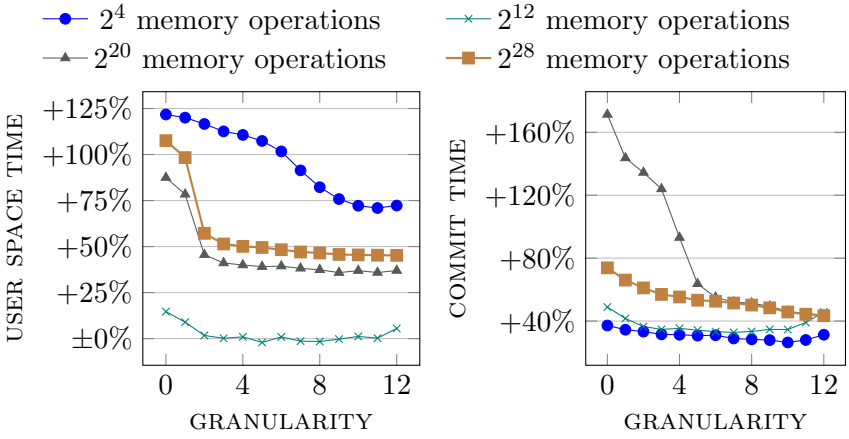
[2]We divide the average runtime per task by the sum of the average number of read pages and written pages. This is 4 µs per page for *Heat*, 11 µs for *Fft*, 34 µs for *Cilksort* and ≥216 µs for all others.

which amortizes better if larger regions of the page are actually read or modified.

6.3.3 Effect of Additional Instrumentation

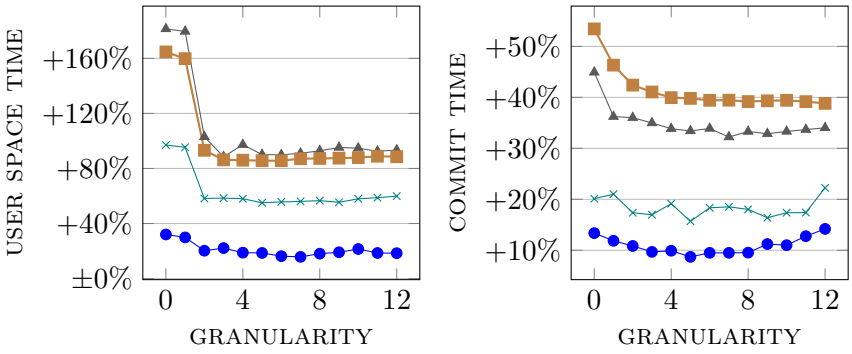
In order to evaluate the effectiveness and performance of the K-TLS$^+$ approach, we first run micro benchmarks to assess the impact on the individual phases of execution, in particular the execution of the instrumented program in user space, the validation phase in kernel space and the commit phase in kernel space. We expect all of those phases to be slowed down by the additional management of the shadow data. We run both of the micro benchmarks introduced in Section 5.3.1.1 and also used in Section 6.3.1. Then, we run the *Cilk* programs parallelized in Sambamba in order to evaluate the impact on automatically parallelized real-world programs.

The micro benchmarks perform random or linear writes to disjoint memory blocks (see Section 5.3.1.2). We measure the times for user space execution, validation and commit for all possible granularities (from 2^0 to 2^{12}) and four different workloads. The results are shown in Figure 6.5. For the validation time, no increase is measurable, so we skip that plot. As described in Section 6.2.4, validation only examines the shadow memory if a conflict on a memory page was detected. As these benchmarks write to disjoint memory regions, this filter already proves all tasks disjoint, hence the runtime is exactly the same as for K-TLS. For the user-space execution time plotted in 6.5a and 6.5c, we see that the overhead generally decreases towards coarser granularities. This is expected, as less shadow memory is accessed, hence the memory caches in hardware can be utilized better.



(A) User space execution time increase compared to K-TLS for random writes to a fixed memory block

(B) Increase in the commit time compared to K-TLS for random writes to a fixed memory block



(C) User space execution time increase compared to K-TLS for linearly writing a memory block

(D) Increase in the commit time compared to K-TLS for linearly writing a memory block

FIGURE 6.5: Overhead introduced by K-TLS$^+$ for tracking memory accesses during execution (A and C) and using it during commit (B and D). We use the same benchmarks as introduced in Section 5.3.1.2. The execution time decreases if the granularity increases, but between 50 % and 100 % degradation must be expected. As commit only takes a small fraction of the overall execution time, the increase on those numbers is not problematic. For validation, the increase is not measurable.

Remember that the instrumented code for different granularities only differs by some constants, except for very small granularities which require multi-MDU-updates (see Section 6.2.3.2). For random updates (Figure 6.5a), there is the interesting case of 2^{12} memory operations, where the increase in execution time is around 0 % for most granularities. In order to verify those measurements, we ran another test with a fixed granularity of 4, and all powers of two between 2^4 and 2^{20}, because at these boundaries, the plot shows substantial slowdowns. Indeed, the overhead decreases between 2^4 and 2^8 operations, then stays around 0 % until 2^{14}, and then increases again. The program code executed for these tests is always the same, we just vary one input value. We profiled the program for inputs with no measurable overhead, and detected that even though additional memory operations are performed to update the shadow memory, they do not cause any delay during execution. They seem to perfectly hide in the stalls produced by the original memory updates.

The commit time is increased compared to K-TLS for two reasons: First, for each modified memory the shadow memory needs to be examined, and second, the memory content is copied to the respective page in the parent process instead of replacing the page in the page table (see Section 6.2.4). For random memory accesses (Figure 6.5b) the commit time increases more than for linear accesses (Figure 6.5d), especially for small or medium workloads. The commit procedure prefers continuous modified memory regions, as they are copied in one single *memcpy* operation, and also the examination of the shadow memory is faster in these cases.

In the next experiment, we re-run the automatically parallelized *Cilk* programs with K-TLS$^+$. Table 6.3 on page 156 shows the number

TABLE 6.3: Static number of *meta data unit* (MDU) updates for the benchmark programs from the *Cilk* suite. For the different granularities, we list the number of *single-MDU updates*, *reduced multi-MDU updates* that touch at most two memory words, and full *multi-MDU updates*. At the lowest granularity, most memory accesses have to be instrumented for reduced multi-MDU updates, since they touch more than one byte of memory. With coarser granularities, most locations become single-MDU updates. The remaining locations are either not sufficiently aligned, or update a dynamically sized memory block.

Benchmark	static number of MDU updates		
	gran. 0 – 1	gran. 2	gran. 3 – 12
Cilksort	0 / 66 / 6	0 / 66 / 6	66 / 0 / 6
Fft	0 / 891 / 0	691 / 200 / 0	757 / 134 / 0
Heat	0–1 / 85–86 / 0	30 / 56 / 0	86 / 0 / 0
Lu	0 / 85 / 0	40 / 45 / 0	85 / 0 / 0
Matmul	0 / 66 / 0	48 / 18 / 0	66 / 0 / 0
Plu	0 / 119 / 0	105 / 14 / 0	119 / 0 / 0
Spacemul	0 / 312 / 0	4 / 308 / 0	312 / 0 / 0
Strassen	0 / 114 / 0	39 / 75 / 0	114 / 0 / 0

of memory operations that had were automatically instrumented in each program. It also lists which of these operations required a *single-MDU update*, a *reduced multi-MDU update* or a *full multi-MDU update* (see Section 6.2.3 for details). The latter were only needed in the *Cilksort* program for instrumenting *memcpy* calls with input-dependent sizes. All of the other instrumented locations are ordinary loads and stores with statically known sizes, and *memcpy* invocations with statically known sizes. All of those could be represented either by single-MDU updates or by reduced multi-MDU updates. This means that none of them writes more than 33 bytes. In fact, the vast

majority accesses structures of either 4 or 8 bytes, corresponding to ordinary *float*, *double*, *int* or *long* values. At a granularity of 2 or 3 (i.e. 4 or 8 bytes), those accesses turn in single-MDU updates, such that at all granularities greater or equal to 3, only a small number of reduced multi-MDU updates remain. As seen in the table, they only remain for the *Fft* program, where they are generated to copy structs of two *float* values. These structs have a size of 8 bytes, but only an alignment of 4 bytes, which potentially requires updating multiple MDUs for all granularities.

Figure 6.6 shows the result of executing the Cilk programs in K-TLS$^+$. K-TLS was able to speed up seven out of the eight programs, with more than 2× speedup on six of them. K-TLS$^+$ however only speeds up three to five of them, depending on the granularity level. In order to illustrate the reason for this, we also plot the user-space execution time, which is the run time of the instrumented program. This is plotted in Figure 6.7. We see that the overhead relative to K-TLS is often much larger than in the micro benchmarks we ran before. Profiling this reveals that instead of executing roughly twice the amount of memory operations, these programs execute 5 to 6 times the number of memory operations. Looking into the profile for *Spacemul* reveals that most of the time is spent on the kernel function which does the matrix multiplication for 16×16 blocks. Its code size increases by a factor of 3.2, and the size of the allocated stack slot by a factor of 5.5. This is because the additional instructions inserted during instrumentation require several registers during execution, so many other values have to be spilled to the stack. In these large functions operating repeatedly on dozens of memory cells, reloading many more values either from the stack or the heap is required,

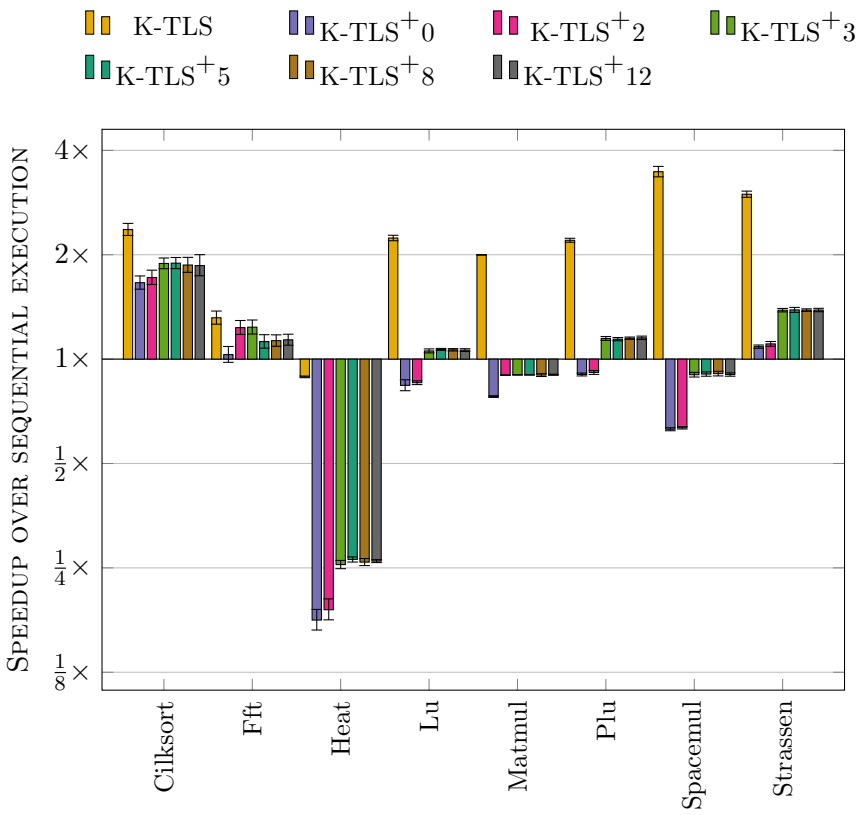


FIGURE 6.6: The speedup of K-TLS$^+$ with different granularities over sequential execution on Cilk programs without memory conflicts. For most applications, the overhead of tracking memory updates does not pay off. Only for five out of the eight programs, we still get speedups over sequential execution.

resulting in the large overheads observed.

In order to evaluate the effectiveness in reducing the number of false conflicts reported, we took the *Cilksort* program, for which K-TLS$^+$ shows moderate overheads, and ran it with different sizes, such that the memory accesses of the parallel tasks were not located on disjoint pages. Figure 6.8 shows the result of this experiment.

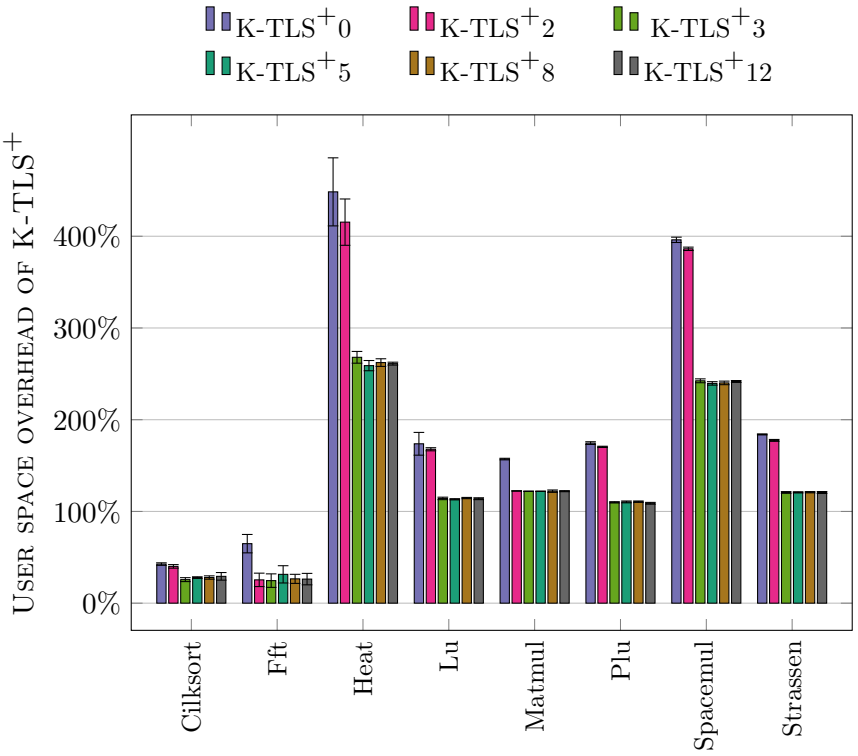


FIGURE 6.7: Overhead caused by tracking memory updates in user space at different granularities, relative to execution in K-TLS. In many applications, this overhead is much larger than in our micro benchmarks, and is often caused more by the code bloat and increased register pressure than the memory tracking itself. Only the *Cilksort* and *Fft* programs show slowdowns in the expected range.

The first size of 2^{22} is the baseline. In this configuration, the four quarters which are sorted in parallel will all be on disjoint pages, so all configurations achieve a good speedup here. If the size is decreased by 1024, the coarsest granularity of 2^{12} (the size of a page) already generates three false conflicts on the eight tasks which are executed. For K-TLS, this is still enough to provide a little speedup,

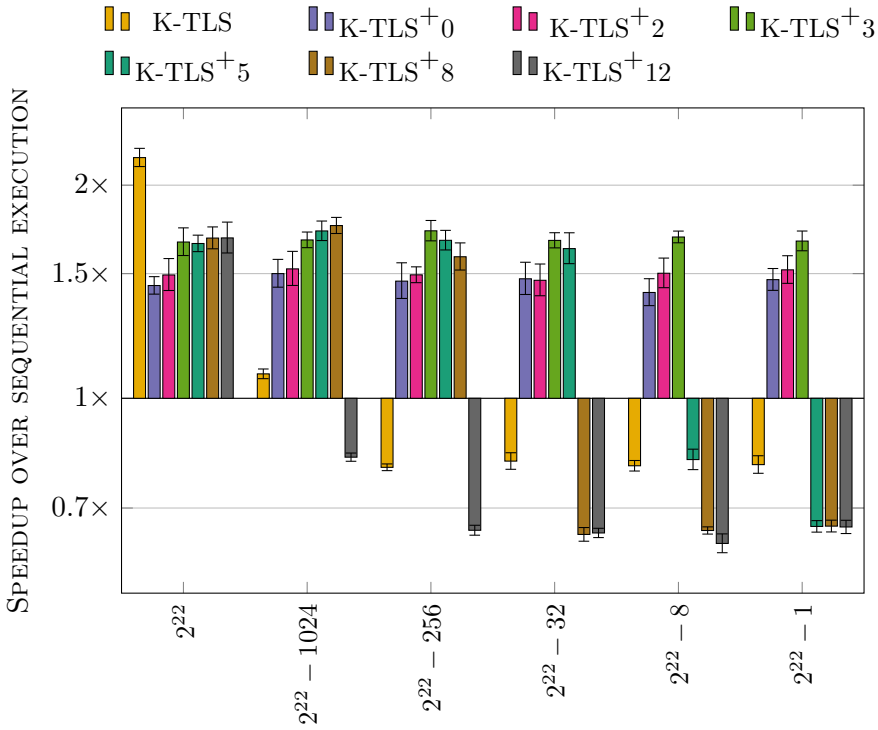


FIGURE 6.8: Performance of *Cilksort* for different sizes, executed in K-TLS or K-TLS$^+$ with different granularities. We report speedup over sequential execution. Finer granularities still achieve speedup for less aligned array sizes.

for K-TLS$^+$ with page-size granularity, we already see a slowdown. For a decrease of 256, five tasks show conflicts and need to be re-executed, resulting in a slowdown for both K-TLS and K-TLS$^+$ with page-size granularity. If we just reduce the size by 32, 8, or 1 element, the four quarters of the array will be less and less aligned, resulting in a finer granularity which is necessary to still detect that no conflict has happened. Since *Cilksort* operates on machine words of 64 bit (or 8 byte), all granularities between 2^0 and 2^3 are sufficient to eliminate

all false conflicts[3]. Hence they all provide speedup independent of the alignment.

6.4 Conclusion

In this chapter we presented two *novel virtual-memory based* systems for thread level speculation, K-TLS and K-TLS$^+$. Both implement major parts of their functionality directly in the operating system. We demonstrated that this speeds up all operations substantially in comparison to user-space only approaches. By spawning tasks faster than the U-TLS system, we are able to successfully apply speculative parallelization to tasks with execution times between a few milliseconds up to several minutes and get performance improvements for this full range. TinySTM$^+$ fails to provide such speedups if the tasks access too many memory locations. User-space implementations suffer from large spawn cost per speculative task and from expensive copying of memory during commit. K-TLS solves both of these problems and provides additional safety guarantees which no user-space solution provides. It isolates the speculative tasks even in the presence of system calls and thus allows to call any—potentially unknown or untrusted—code from speculative tasks.

The major drawback of virtual-memory based approaches is the granularity of access tracking. Traditionally, conflict detection is only performed on whole memory pages of 4 kB. In order to solve this problem, we proposed to combine the kernel-based K-TLS approach with instrumentation. This poses further restrictions on the code which

[3] assuming that the start address of the array is aligned to at least 8 byte, which is the case in our implementation

can be executed inside of transactions, in particular the *intermediate representation* of all code must be available for instrumentation. Thus, no *external functions* can be called. This condition is dynamic—if calls to external code exist, they only trigger a rollback once they are executed. Also, parallel sections executing instrumented code can be mixed with uninstrumented code within the same program—although all tasks which belong to the same dynamic instance of a parallel section have to agree on the same technique. One could even switch between different variants—uninstrumented, or instrumented for various granularities—for the same parallel section based on runtime information. Such dynamic adaptions are out of the scope of this dissertation though. Whether the K-TLS$^+$ system—combining K-TLS with instrumentation—still provides a satisfying performance depends on the characteristics of the code. We have shown that while the overhead is often less than 50 % or even close to 0 %, it can also go up to $5 - 6\times$.

Chapter 7

Conclusions and Future Work

In order to leverage the power of modern multi-core or many-core machines, different automatic parallelization schemes were proposed. The static analyses used to find parallelization opportunities, however, often have to fall back to overapproximations on complex programs, leading to pessimistic parallelization decisions. Many authors found this to be a problem and propose to use *speculation* to allow for *optimistic* assumptions. As these might be unsafe in general, a mechanism is needed to detect at runtime whether the assumptions hold. For some occurrences it is possible to prove the assumptions before starting parallel computation. In the general case however, runtime techniques are needed which check for *misspeculations* while the parallel program is being executed. Some authors just assume the availability of such a system in hardware and simulate its execution with unverifiable performance assumptions. Others come up with ad hoc implementations which are neither described in detail nor publicly

TABLE 7.1: Characteristics of the different speculative runtime systems examined in this thesis. STM resembles the state of the art in explicit memory tracking. U-TLS implements virtual-memory based speculation as proposed by different authors. K-TLS is novel by utilizing a Linux kernel module. K-TLS$^+$ builds on K-TLS but adds instrumentation to increase the granularity of memory tracking.

	pure userspace	instrumentation	OS support	efficient setup	efficient execution	efficient commit	finegranular
STM	+	+	–	+	–	–	+
U-TLS	+	–	0	–	+	0	–
K-TLS	–	–	+	–	+	+	–
K-TLS$^+$	–	+	+	–	0	+	+

available. This makes it difficult to assess their generalizability or reuse them for further research. The lack of a reusable and general speculation system ties up resources that could otherwise be spent on comping up and experimenting with more sophisticated speculative parallelization schemes. Also, those schemes would be better comparable with each other if they were using the same established speculation system.

This thesis thus focussed on the development of a general, reusable and easy-to-use runtime system for thread level speculation, providing the best possible performance for a wide range of applications. To this end, we made the following contributions:

- We evaluated the performance of the state of the art STM system TinySTM on different benchmarks, and discovered severe *performance problems* when applying it to automatically parallelized programs. We *highlighted the differences* in the characteristics of these applications against the typical uses of STM. Based on these findings, we proposed several *changes to the implementation*, and evaluated the runtime impact of these improvements. Even though the runtime improves by orders of magnitude, the explicit memory tracking of STM still involves too high overheads to be usable in the context of automatic parallelization.

- We designed a *virtual-memory based* runtime system for thread level speculation along the lines of designs proposed by different authors. In contrast to related work, we fully describe the design and implementation of this system called U-TLS and make it available as open source. We evaluated its performance on automatically parallelized programs and found it to *perform significantly better* than STM. For some of the programs, U-TLS provides remarkable speedups. However, we notice some drawbacks which cannot be avoided by a pure user-space implementation.

- Based on the experience we made with U-TLS, we designed and implemented a *kernel-based* runtime system for thread-level speculation called K-TLS. It *significantly improves the runtime* of all TLS operations: task creation, execution, validation and commit. Furthermore, it provides *stronger isolation guarantees* than any user-space implementation can ever provide: It intercepts all system calls and triggers a rollback for each call

that has a potential side effect on the process. This allows to call any code from a speculative task, even if it is unknown or untrusted.

- K-TLS$^+$ is a combination of the K-TLS system and code instrumentation. By allocating a shadow memory region and instrumenting all memory operations to set a respective bit in this shadow memory, we have much more fine-grained information about the accessed memory regions of each task. We use this information to eliminate false conflicts during validation. The granularity can be chosen between 1 byte and the size of a memory page (4096 bytes). We demonstrated that this approach effectively eliminates false conflicts. Its overhead over K-TLS is sometimes negligible, but it can also increase to several times the original execution time.

7.1 U-TLS

The source code of U-TLS is available on GitHub, in the common repository for all the runtime systems developed for this thesis:

`https://github.com/hammacher/k-tls`

The major part of the implementation of U-TLS is found in the file `lib/TLS/utls.cpp`. It has a length of about 1000 lines of code, and uses common functionality like the TLS task list implemented in other files.

There are several ways to further improve the performance of U-TLS. Some of the ideas are listed in the following:

Reuse processes. Similar to K-TLS, one could try to reuse the processes of finished tasks to execute later tasks. There are some challenges though: First, one would need to reset the page table of the process before reusing it. There is no system call to do this for general memory mappings, so in order to clear the page table entries, the whole mapping would need to be removed using the *munmap* system call. But then re-creating those mappings is not possible for anonymous mappings. Therefore the whole memory management strategy of U-TLS would need to change, such that the memory content is copied from the parent process on demand. As this includes higher cost per page fault, this whole change would probably only pay off for rather small tasks which are dominated by the process forking cost.

Improve commit time by using shared memory. In order to reduce the commit time, one could avoid the copy operation involved by using shared memory. The child process would not write its changes to anonymous pages and transfer them to the parent during commit, but would write to pages shared between both processes. One option would be to use *named shared memory*, and make the parent process map the individual changed pages to its own address space during commit. The other option is to use a preallocated anonymous shared mapping, and make the child process remap individual pages into its address space as needed. Instead of communicating page contents to the parent during commit, it would then only communicate which pages were mapped where, and the parent would apply the same mapping. It has been found by others

however that remapping individual pages takes several times longer than copying the content of that page, so this method would probably only pay off if larger continuous regions are written by a task. Both of these implementations (named or anonymous shared memory), however, would leave the virtual memory view of the parent process changed after executing speculative tasks, so they are way more intrusive than the current implementation.

Recursive speculative parallelization. In order to extract scalable parallelism from divide-and-conquer algorithms, but also for other applications, it would be beneficial to parallelize recursively. This would allow to spawn speculative sub-tasks from speculative tasks. The child process executing the recursive spawn would then act as parent process to its children, forming a tree of processes. During commit, the changes would be copied into its own address space, and the respective pages would be marked as changed. If a conflict is detected, just that sub-child would re-execute. If one task is rolled back by discarding its memory changes, also the effect of all sub-tasks is discarded, hence they need to re-execute later. This scheme is known as *closed nesting*.

7.2 K-TLS

Also the source code of K-TLS is available in the same repository on GitHub:

`https://github.com/hammacher/k-tls`

The kernel module is implemented in the `lib/KTLS` directory, and consists of more than 3000 lines of code. It is compiled by the included Makefile against the currently running kernel. The slim user-space interface is implemented in `lib/TLS/ktls.cpp`. The *cindex* benchmark used in Section 4.3.3 is available in the *cindex-benchmark* branch.

Also for K-TLS, we have a few ideas of what could be improved further:

Replay system calls. The current implementation allows a small number of system calls for which we know that they have no side effect. We could add support for some of the remaining system calls by replaying them in the parent process later. It it is a system call with an effect shared between both processes, e.g. file output, we just do not run it in the child, and return a speculative value signalling success. When the actual system call replayed in the parent process later returns a different value, we would not commit any of the memory changes of the child, and re-execute it while omitting any system call which was already replayed. This would require the speculative task to be deterministic, however, such that the same sequence of system calls is generated. For system calls which only have an effect on the process executing them, like mapping new memory regions, we would execute them both in the child and in the parent process.

Recursive speculative parallelization. Just like for U-TLS, also in K-TLS we could add support for recursive task spawning. We would need to add special handling for the *ioctl* system call

which is used to communicate with the kernel module. The semantics of recursively spawned tasks would then be the same as described for U-TLS.

More fine-granular page versioning. In its current implementation, each modified page carries a *version* representing the id of the last task which modified this page. Each task stores a *start version*, which is the id of the last committed task at the moment when the current task reads the first page. A conflict is detected if at commit time *any read page* has a higher version than the start version of the task. Some conflicts can potentially be avoided by storing the version number of each read page instead of one version number per task. This would especially make a difference if the tasks are unbalanced, such that some task commits early and other tasks later read the page modified by the earlier task. We already deliver the last committed page in the parent whenever a child process first accesses a page, but currently conservatively assign an older version id to it by only storing one version per task.

7.3 Sambamba

The Sambamba framework contains the work of different authors, and many parts are unstable or unusable in its current form. We therefore decided to not publish the source code of Sambamba yet. Interested research groups however get access to the source code with a respective disclaimer.

In order for Sambamba to effectively support speculation, a few changes are needed:

Adaptive switching between runtime systems. As shown in the individual evaluation sections, there is no single runtime system for thread level speculation which provides the best performance in all cases. Especially deciding which granularity for K-TLS^{+} performs best is hard to do a priori. Sambamba already supports dynamically generating the parallel code per section for any of the presented runtime systems. What is missing though is a feedback mechanism which iteratively tries to find the best runtime system per section, potentially even considering the input values which determine the behaviour of the parallel tasks.

Include speculation in ILP-based parallelization. Sambamba uses *integer linear programming (ILP)* in order to find the optimal parallel schedule per set of basic blocks with the same control dependencies. In order to better detect speculative parallelization opportunities, this ILP formulation would need to be extended such that it is able to ignore certain dependencies. Information from previous runs can be used to make a more qualified decision here.

Bibliography

[1] Advanced Micro Devices (AMD). *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. `http://developer.amd.com/wordpress/media/2012/10/31116.pdf`. [PDF, accessed 11-Mar-2016]. Apr. 2010.

[2] Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris Kirkham, Mikel Luján, and Kim Jarvis. "Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory". In: *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing - ICA3PP '08*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 196–207. ISBN: 9783540695004. DOI: `10.1007/978-3-540-69501-1_21`.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '09*. New York, New York, USA: ACM Press, 2009, pages 38–49. ISBN: 9781605583921. DOI: `10.1145/1542476.1542481`.

[4] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. "Grace: Safe Multithreaded Programming for C/C++". In: *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09*. New York, New York, USA: ACM Press, 2009,

pages 81–96. ISBN: 9781605587660. DOI: 10.1145/1640089.1640096.

[5] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (Oct. 1966), pages 757–763. ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565.

[6] Anasua Bhowmik and Manoj Franklin. "A general compiler framework for speculative multithreading". In: *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures - SPAA '02*. New York, New York, USA: ACM Press, Aug. 2002, pages 99–108. ISBN: 1581135297. DOI: 10.1145/564870.564885.

[7] Daniel Birtel. "Variable Granularity TLS via Code Instrumentation". Bachelor's Thesis. Saarland University, July 2015.

[8] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13 (1970), pages 422–426. ISSN: 00010782. DOI: 10.1145/362686.362692.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '95*. New York, New York, USA: ACM Press, 1995, pages 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and*

Implementation - PLDI '08. New York, New York, USA: ACM Press, 2008, pages 101–113. ISBN: 9781595938602. DOI: 10.1145/1375581.1375595.

[11] Derek Bruening and Qin Zhao. "Practical memory checking with Dr. Memory". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '11*. IEEE, Apr. 2011, pages 213–223. ISBN: 978-1-61284-356-8. DOI: 10.1109/CGO.2011.5764689.

[12] Mihai Burcea, J. Gregory Steffan, and Cristiana Amza. "The Potential for Variable-Granularity Access Tracking for Optimistic Parallelism". In: *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness - MSPC '08*. New York, New York, USA: ACM Press, 2008, pages 11–15. ISBN: 9781605580494. DOI: 10.1145/1353522.1353527.

[13] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. "Automatic generation of nested, fork-join parallelism". In: *The Journal of Supercomputing* 3.2 (July 1989), pages 71–88. ISSN: 0920-8542. DOI: 10.1007/BF00129843.

[14] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. "Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation - PACT '14*. 2014, pages 187–200. ISBN: 9781450328098. DOI: 10.1145/2628071.2628086.

[15] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. "HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs". In: *41st ACM/IEEE International Symposium on Computer Architecture - ISCA'14*. IEEE, June 2014, pages 217–228. ISBN: 978-1-4799-4394-4. DOI: 10.1109/ISCA.2014.6853215.

[16] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. "HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization - CGO '12*. 2012, pages 84–93. ISBN: 9781450312066. DOI: 10.1145/2259016.2259028.

[17] Luis Ceze, James Tuck, Călin Caşcaval, and Josep Torrellas. "Bulk Disambiguation of Speculative Threads in Multiprocessors". In: *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 2006, pages 227–238. ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.13.

[18] Michael K. Chen and Kunle Olukotun. "The Jrpm system for dynamically parallelizing Java programs". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* June. IEEE Comput. Soc, 2003, pages 434–445. ISBN: 0-7695-1945-8. DOI: 10.1109/ISCA.2003.1207020.

[19] Marcelo Cintra and Diego R. Llanos. "Toward efficient and robust software speculative parallelization on multiprocessors". In: *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '03.*

New York, New York, USA: ACM Press, 2003, pages 13–24. ISBN: 1581135882. DOI: 10.1145/781498.781501.

[20] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. "Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory". In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11*. 2011, pages 39–52. ISBN: 9781450302661. DOI: 10.1145/2248487.1950373.

[21] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. "NOrec: streamlining STM by abolishing ownership records". In: *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10)*. 2010, pages 67–78. ISBN: 9781605587080. DOI: 10.1145/1693453.1693464.

[22] Alain Darte, Georges-André Silber, and Frédéric Vivien. "Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling". In: *Parallel Processing Letters* 7.4 (1996), pages 379–392.

[23] Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim. "Runtime parallelization of legacy code on a transactional memory system". In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers - HiPEAC '11*. New York, New York, USA: ACM Press, 2011, pages 127–136. ISBN: 9781450302418. DOI: 10.1145/1944862.1944882.

[24] Dave Dice, Ori Shalev, and Nir Shavit. "Transactional Locking II". In: *Proceedings of the 20th International Conference on Distributed Computing - DISC '06*. 2006, pages 194–208. DOI: 10.1007/11864219_14.

[25] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. "Software behavior oriented parallelization". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '07*. New York, New York, USA: ACM Press, 2007, pages 223–234. ISBN: 9781595936332. DOI: 10.1145/1250734.1250760.

[26] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. "Why STM can be more than a research toy". In: *Communications of the ACM* 54.4 (Apr. 2011), pages 70–77. ISSN: 00010782. DOI: 10.1145/1924421.1924440.

[27] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. "Stretching transactional memory". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '09*. May 2009, pages 155–165. ISBN: 978-1-60558-392-1. DOI: 10.1145/1543135.1542494.

[28] Paul Feautrier. "Automatic Parallelization in the Polytope Model". In: *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Lecture Notes in Computer Science 1132 (1996). Edited by Guy-René Perrin and Alain Darte, pages 79–103.

[29] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International Journal of Parallel Programming* 21.5 (Oct. 1992), pages 313–347. ISSN: 0885-7458. DOI: 10.1007/BF01407835.

[30] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pages 389–420. ISSN: 0885-7458. DOI: 10.1007/BF01379404.

[31] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. "Time-Based Software Transactional Memory". In: *IEEE Transactions on Parallel and Distributed Systems* 21.12 (Dec. 2010), pages 1793–1807. ISSN: 1045-9219. DOI: 10.1109/TPDS.2010.49.

[32] Pascal Felber, Christof Fetzer, and Torvald Riegel. "Dynamic performance tuning of word-based software transactional memory". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '08*. New York, New York, USA: ACM Press, 2008, pages 237–245. ISBN: 9781595937957. DOI: 10.1145/1345206.1345241.

[33] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pages 319–349. ISSN: 01640925. DOI: 10.1145/24039.24041.

[34] B. Fleisch and G. Popek. "Mirage: a coherent distributed shared memory design". In: *Proceedings of the 12th ACM*

Symposium on Operating Systems Principles - SOSP '89. 1989, pages 211–223. DOI: 10.1145/74851.74871.

[35] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.2 (Apr. 1983), pages 164–189. ISSN: 01640925. DOI: 10.1145/69624.357206.

[36] Justin Gottschlich, Manish Vachharajani, and Jeremy Siek. "An efficient software transactional memory using commit-time invalidation". In: *Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '10*. 2010, pages 101–110. ISBN: 9781605586359. DOI: 10.1145/1772954.1772970.

[37] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. "STMBench7: A Benchmark for Software Transactional Memory". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems - EuroSys '07*. 2007, pages 315–324. ISBN: 9781595936363. DOI: 10.1145/1272996.1273029.

[38] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. "Maximizing multiprocessor performance with the SUIF compiler". In: *IEEE Computer* 29.12 (Dec. 1996), pages 84–89. ISSN: 0018-9162. DOI: 10.1109/2.546613.

[39] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. "The Stanford Hydra CMP". In: *IEEE Micro* 20.2 (2000), pages 71–84. ISSN: 02721732. DOI: 10.1109/40.848474.

[40] Tim Harris and Keir Fraser. "Language support for lightweight transactions". In: *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications - OOPSLA '03*. New York, New York, USA: ACM Press, 2003, pages 388–402. ISBN: 1581137125. DOI: 10.1145/949305.949340.

[41] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. "Optimizing memory transactions". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '06*. ACM, June 2006, pages 14–25. DOI: 10.1145/1133255.1133984.

[42] Maurice Herlihy. "A methodology for implementing highly concurrent data structures". In: *ACM SIGPLAN Notices* (Mar. 1990), pages 197–206. ISSN: 03621340. DOI: 10.1145/99164.99185.

[43] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. "Software transactional memory for dynamic-sized data structures". In: *Proceedings of the twenty-second annual symposium on Principles of distributed computing - PODC '03*. 2003, pages 92–101. ISBN: 1581137087. DOI: 10.1145/872035.872048.

[44] Maurice Herlihy and J. Eliot B. Moss. "Transactional memory: Architectural support for lock-free data structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture - ISCA '93*. ACM, 1993, pages 289–300. DOI: 10.1145/173682.165164.

[45] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. "Hopscotch hashing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Volume 5218 LNCS. 2008, pages 350–364. ISBN: 3540877789. DOI: 10.1007/978-3-540-87779-0_24.

[46] Maurice Herlihy and Jeannette M Wing. "Axioms for concurrent objects". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages - POPL '87*. New York, New York, USA: ACM Press, 1987, pages 13–26. ISBN: 0897912152. DOI: 10.1145/41625.41627.

[47] Ben Hertzberg and Kunle Olukotun. "Runtime automatic speculative parallelization". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '11*. IEEE, Apr. 2011, pages 64–73. ISBN: 978-1-61284-356-8. DOI: 10.1109/CGO.2011.5764675.

[48] Antony Hosking, J. Eliot B. Moss, and Darko Stefanovic. "A comparative performance evaluation of write barrier implementation". In: *ACM SIGPLAN Notices* 27.10 (Oct. 1992), pages 92–109. ISSN: 03621340. DOI: 10.1145/141937.141946.

[49] Jialu Huang, Thomas B. Jablin, Stephen R. Beard, Nick P. Johnson, and David I. August. "Automatically exploiting cross-invocation parallelism using runtime information". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization - CGO '13*. IEEE, Feb. 2013, pages 1–11. ISBN: 978-1-4673-5525-4. DOI: 10.1109/CGO.2013.6495001.

[50] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. "Decoupled software pipelining creates parallelization opportunities". In: *Proceedings of the 8th IEEE/ ACM International Symposium on Code Generation and Optimization - CGO '10*. 2010, pages 121–130. ISBN: 9781605586359. DOI: 10.1145/1772954.1772973.

[51] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. "Speculative thread decomposition through empirical optimization". In: *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '07*. New York, New York, USA, 2007, pages 205–214. ISBN: 9781595936028. DOI: 10.1145/1229428.1229474.

[52] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems". In: *Proceedings of the 1994 Winter Technical Conference on USENIX - WTEC '14*. 1994, pages 115–131.

[53] Kirk Kelsey, Chengliang Zhang, and Chen Ding. "Fast Track: Supporting Unsafe Optimizations with Software Speculation". In: *16th International Conference on Parallel Architecture and Compilation Techniques - PACT '07*. IEEE, Sept. 2007, pages 414–429. ISBN: 0-7695-2944-5. DOI: 10.1109/PACT.2007.4336242.

[54] Gokcen Kestor, Roberto Gioiosa, Tim Harris, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. "STM2: A parallel STM for high performance simultaneous multithreading systems". In: *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*

- *PACT '11*. 2011, pages 221–231. ISBN: 9780769545660. DOI: 10.1109/PACT.2011.54.

[55] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. "Automatic speculative DOALL for clusters". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization - CGO '12*. New York, New York, USA: ACM Press, 2012, pages 94–103. ISBN: 9781450312066. DOI: 10.1145/2259016.2259029.

[56] Andi Kleen. *Virtual memory map with 4 level page tables*. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt. [Online; accessed 11-Mar-2016]. July 2004.

[57] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. "Optimistic Parallelism Requires Abstractions". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '07*. New York, New York, USA: ACM Press, 2007, pages 211–222. ISBN: 9781595936332. DOI: 10.1145/1250734.1250759.

[58] Leslie Lamport. "Specifying Concurrent Program Modules". In: *ACM Transactions on Programming Languages and Systems* 5.2 (Apr. 1983), pages 190–222. ISSN: 01640925. DOI: 10.1145/69624.357207.

[59] Chris Lattner, Andrew Lenharth, and Vikram Adve. "Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World". In: *ACM SIGPLAN Notices*. Volume 42. 6. ACM, 2007, pages 278–289. DOI: 10.1145/1273442.1250766.

[60] Christian Lengauer. "Loop Parallelization in the Polytope Model". In: *4th International Conference on Concurrency Theory - CONCUR'93*. Hildesheim, Germany: Springer-Verlag, 1993, pages 398–416. ISBN: 3-540-57208-2. DOI: 10.1007/3-540-57208-2_28.

[61] Kai Li and Paul Hudak. "Memory coherence in shared virtual memory systems". In: *ACM Transactions on Computer Systems (TOCS)* 7.4 (1989), pages 321–359. ISSN: 07342071. DOI: 10.1145/75104.75105.

[62] Amy W. Lim and Monica S. Lam. "Maximizing parallelism and minimizing synchronization with affine transforms". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '97*. New York, New York, USA: ACM Press, 1997, pages 201–214. ISBN: 0897918533. DOI: 10.1145/263699.263719.

[63] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. "POSH: A TLS Compiler that Exploits Program Structure". In: *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '06*. New York, New York, USA: ACM Press, Mar. 2006, pages 158–167. ISBN: 1595931899. DOI: 10.1145/1122971.1122997.

[64] Sandya S. Mannarswamy and Ramaswamy Govindarajan. "Variable Granularity Access Tracking Scheme for Improving the Performance of Software Transactional Memory". In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium - IPDPS '11*. IEEE, May 2011,

pages 455–466. ISBN: 978-1-61284-372-8. DOI: 10.1109/IPDPS.2011.51.

[65] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2008. ISBN: 978-0470343432.

[66] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '09*. New York, New York, USA: ACM Press, 2009, pages 166–176. ISBN: 9781605583921. DOI: 10.1145/1542476.1542495.

[67] Chí Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. "STAMP: Stanford Transactional Applications for Multi-Processing". In: *Proceedings of the 4th IEEE International Symposium on Workload Characterization - IISWC '08*. IEEE, Oct. 2008, pages 35–46. ISBN: 978-1-4244-2777-2. DOI: 10.1109/IISWC.2008.4636089.

[68] Vijay Nagarajan and Rajiv Gupta. "Architectural support for shadow memory in multiprocessors". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE '09*. New York, New York, USA: ACM Press, 2009, pages 1–10. ISBN: 9781605583754. DOI: 10.1145/1508293.1508295.

[69] Nicholas Nethercote and Julian Seward. "How to shadow every byte of memory used by a program". In: *Proceedings of the 3rd International Conference on Virtual Execution Environments - VEE '07*. New York, New York, USA: ACM Press, 2007,

pages 65–74. ISBN: 9781595936301. DOI: 10.1145/1254810.1254820.

[70] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '07*. New York, New York, USA: ACM Press, 2007, pages 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[71] Nicholas Nethercote and Julian Seward. "Valgrind: A Program Supervision Framework". In: *Electronic Notes in Theoretical Computer Science* 89.2 (Oct. 2003), pages 44–66. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)81042-9.

[72] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. "A lightweight in-place implementation for software thread-level speculation". In: *Proceedings of the 21st ACM Annual Symposium on Parallelism in Algorithms and Architectures - SPAA '09*. New York, New York, USA: ACM Press, Aug. 2009, pages 223–232. ISBN: 9781605586069. DOI: 10.1145/1583991.1584050.

[73] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. *Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor*. Technical report. Stanford University, 1997.

[74] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. "Automatic Thread Extraction with Decoupled Software Pipelining". In: *38th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO '05*. IEEE, 2005, pages 105–118. ISBN: 0-7695-2440-0. DOI: 10.1109/MICRO.2005.13.

[75] Spiros Papadimitriou and Todd C. Mowry. *Exploring Thread-Level Speculation in Software: The Effects of Memory Access Tracking Granularity*. Technical report July. 2001.

[76] William Pugh. "The Omega Test: a fast and practical integer programming algorithm for dependence analysis". In: *Proceedings of the 5th International Conference on Supercomputing - ICS '91*. New York, New York, USA: ACM Press, 1991, pages 4–13. ISBN: 0897914597. DOI: 10.1145/125826.125848.

[77] Hari K. Pyla, Calvin Ribbens, and Srinidhi Varadarajan. "Exploiting coarse-grain speculative parallelism". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11*. New York, New York, USA: ACM Press, 2011, pages 555–574. ISBN: 9781450309400. DOI: 10.1145/2048066.2048110.

[78] Ravi Rajwar and James R. Goodman. "Speculative lock elision: enabling highly concurrent multithreaded execution". In: *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture - MICRO '02*. 2002, pages 294–305. ISBN: 0-7695-1369-7. DOI: 10.1109/MICRO.2001.991127.

[79] Ravi Rajwar and James R. Goodman. "Transactional lock-free execution of lock-based programs". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '02*. New York, New York, USA: ACM Press, 2002, pages 5–17. ISBN: 1581135742. DOI: 10.1145/605397.605399.

[80] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. "Committing conflicting transactions in an STM". In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '09*. Volume 44. 2009, pages 163–172. ISBN: 978-1-60558-397-6. DOI: 10.1145/1594835.1504201.

[81] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. "Speculative parallelization using software multi-threaded transactions". In: *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10*. New York, New York, USA: ACM Press, 2010, pages 65–76. ISBN: 9781605588391. DOI: 10.1145/1736020.1736030.

[82] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. "Parcae: A System for Flexible Parallel Execution". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '12*. New York, New York, USA: ACM Press, 2012, pages 133–144. ISBN: 9781450312059. DOI: 10.1145/2254064.2254082.

[83] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. "Parallel-stage decoupled software pipelining". In: *Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization - CGO '08*. New York, New York, USA: ACM Press, Apr. 2008, page 114. ISBN: 9781595939784. DOI: 10.1145/1356058.1356074.

[84] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. "Decoupled software pipelining with the

synchronization array". In: *13th International Conference on Parallel Architectures and Compilation Techniques - PACT '04*. Published by the IEEE Computer Society, 2004, pages 177–188. ISBN: 0-7695-2229-7.

[85] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. "A scalable method for run-time loop parallelization". In: *International Journal of Parallel Programming* 23.6 (Dec. 1995), pages 537–576. ISSN: 0885-7458. DOI: 10.1007/BF02577866.

[86] Lawrence Rauchwerger and David Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '95*. New York, New York, USA: ACM Press, 1995, pages 218–232. ISBN: 0897916972. DOI: 10.1145/207110.207148.

[87] Lawrence Rauchwerger and David Padua. "The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization". In: *Proceedings of the 8th International Conference on Supercomputing - ICS '94*. New York, New York, USA: ACM Press, 1994, pages 33–43. ISBN: 0897916654. DOI: 10.1145/181181.181254.

[88] Lawrence Rauchwerger and David A. Padua. "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization". In: *IEEE Transactions on Parallel and Distributed Systems* 10.2 (1999), pages 160–180. ISSN: 10459219. DOI: 10.1109/71.752782.

[89] Torvald Riegel, Pascal Felber, and Christof Fetzer. "A Lazy Snapshot Algorithm with Eager Validation". In: *20th International Symposium on Distributed Computing - DISC'06*. Edited by Shlomi Dolev. Volume 4167. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 284–298. ISBN: 978-3-540-44624-8. DOI: 10.1007/11864219.

[90] Michael F Ringenburg and Dan Grossman. "AtomCaml: First-class Atomicity via Rollback". In: *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming - ICFP'05*. 2005, pages 92–104. ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086378.

[91] Radu Rugina and Martin Rinard. "Automatic parallelization of divide and conquer algorithms". In: *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '99*. New York, New York, USA: ACM Press, 1999, pages 72–83. ISBN: 1581131003. DOI: 10.1145/301104.301111.

[92] Peter Rundberg and Per Stenström. "An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors". In: *Journal of Instruction-Level Parallelism* 3.2001 (2002).

[93] Karl Rupp. *40 Years of Microprocessor Trend Data*. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/. [Online; accessed 13-Oct-2016]. June 2015.

[94] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. "Sensitivity analysis for automatic parallelization on multi-cores". In: *Proceedings of the 21st annual international conference on Supercomputing - ICS '07*. New York, New York, USA: ACM Press, 2007, pages 263–273. ISBN: 9781595937681. DOI: 10.1145/1274971.1275008.

[95] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. "Hybrid Analysis: Static & Dynamic Memory Reference Analysis". In: *International Journal of Parallel Programming* 31.4 (2003), pages 251–283. ISSN: 08857458. DOI: 10.1023/A:1024597010150.

[96] Mohamed M. Saad, Mohamed Mohamedin, and Binoy Ravindran. "HydraVM: Extracting Parallelism from Legacy Sequential Code Using STM". In: *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism - HotPar'12*. 2012.

[97] Mohamed M. Saad, Roberto Palmieri, and Binoy Ravindran. "Lerna: Transparent and Effective Speculative Loop Parallelization". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Transactional Computing - TRANSACT'16*. 2016.

[98] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. "A Page Coherency Protocol for Popcorn Replicated-kernel Operating System". In: *Proceedings of the ManyCore Architecture Research Community Symposium - MARC*. 2013.

[99] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. "McRT-STM: a high performance software transactional memory system for a multi-core runtime". In: *Proceedings of the eleventh ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '06. New York, New York, USA: ACM, Mar. 2006, pages 187–197. ISBN: 1595931899. DOI: 10.1145/1122971.1123001.

[100] J.H. Saltz, R. Mirchandaney, and K. Crowley. "Run-Time Parallelization and Scheduling of Loops". In: *IEEE Transactions on Computers* 40.5 (1991), pages 603–612. DOI: 10.1145/72935.72967.

[101] Vivek Sarkar. "Automatic partitioning of a program dependence graph into parallel tasks". In: *IBM Journal of Research and Development* 35.5.6 (Sept. 1991), pages 779–804. ISSN: 0018-8646. DOI: 10.1147/rd.355.0779.

[102] Vivek Sarkar. "Determining Average Program Execution Times and their Variance". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '89*. New York, New York, USA: ACM Press, 1989, pages 298–312. ISBN: 089791306X. DOI: 10.1145/73141.74845.

[103] Vivek Sarkar and John Hennessy. "Partitioning Parallel Programs for Macro-Dataflow". In: *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP '86*. New York, New York, USA: ACM Press, 1986, pages 202–211. ISBN: 0897912004. DOI: 10.1145/319838.319863.

[104] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. "Fine-grain Access Control for Distributed Shared Memory". In: *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*

- *ASPLOS '94*. 1994, pages 297–306. ISBN: 0897916603. DOI: 10.1145/381792.195575.

[105] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Presented as part of the 2012 USENIX Annual Technical Conference - ATC '12*. 2012, page 10.

[106] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Proceedings of the fourteenth annual ACM Symposium on Principles of Distributed Computing - PODC '95*. New York, New York, USA: ACM Press, 1995, pages 204–213. ISBN: 0897917103. DOI: 10.1145/224964.224987.

[107] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. "Multiscalar processors". In: *ACM SIGARCH Computer Architecture News* 23.2 (May 1995), pages 414–425. ISSN: 01635964. DOI: 10.1145/225830.224451.

[108] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. "A Scalable Approach to Thread-Level Speculation". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture - ISCA '00*. ACM, 2000, pages 1–12. ISBN: 1-58113-232-8. DOI: 10.1145/342001.339650.

[109] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. "The STAMPede approach to thread-level speculation". In: *ACM Transactions on Computer Systems (TOCS)* 23.3 (Aug. 2005), pages 253–300. ISSN: 07342071. DOI: 10.1145/1082469.1082471.

[110] J. Gregory Steffan and Todd C. Mowry. "The potential for using thread-level data speculation to facilitate automatic parallelization". In: *Proceedings of the 4th International Symposium on High-Performance Computer Architecture - HPCA '98*. IEEE Computer Society, 1998, pages 2–13. ISBN: 0-8186-8323-6. DOI: 10.1109/HPCA.1998.650541.

[111] Janice M. Stone. "A simple and correct shared-queue algorithm using compare-and-swap". In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. 1990, pages 495–504. ISBN: 0-89791-412-0. DOI: 10.1109/SUPERC.1990.130060.

[112] Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. "Generalized Task Parallelism". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.1 (Apr. 2015), 8:1–8:25. ISSN: 15443566. DOI: 10.1145/2723164.

[113] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. "Sambamba : A Runtime System for Online Adaptive Parallelization". In: *Proceedings of the 21st International Conference on Compiler Construction - CC '12*. 2012, pages 240–243. DOI: 10.1007/978-3-642-28652-0_13.

[114] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. "Sambamba : Runtime Adaptive Parallel Execution". In: *Proceedings of the 3rd International Workshop on Adaptive Self-tuning Computing Systems - ADAPT '13*. 2013. ISBN: 9781450320221.

[115] *Threading Building Blocks (Intel® TBB)*. https://www.threadingbuildingblocks.org/. [Online; accessed 13-Feb-2015].

[116] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. "Copy or Discard execution model for speculative parallelization on multicores". In: *41st IEEE/ACM International Symposium on Microarchitecture - MICRO '08*. IEEE, Nov. 2008, pages 330–341. ISBN: 978-1-4244-2836-6. DOI: 10.1109/MICRO.2008.4771802.

[117] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. "Speculative Decoupled Software Pipelining". In: *16th International Conference on Parallel Architecture and Compilation Techniques - PACT '07*. IEEE Computer Society, 2007, pages 49–59. ISBN: 0-7695-2944-5.

[118] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The Paralax Infrastructure: Automatic ParallelizationWith a Helping Hand". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques - PACT '10*. 2010, pages 389–399. ISBN: 9781450301787. DOI: 10.1145/1854273.1854322.

[119] T. N. Vijaykumar and Gurindar S. Sohi. "Task selection for a multiscalar processor". In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture - MICRO '98*. IEEE Comput. Soc, 1998, pages 81–92. ISBN: 0-8186-8609-X. DOI: 10.1109/MICRO.1998.742771.

[120] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. "Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language". In: *Proceedings of the International Symposium on Code Generation and Optimization - CGO '07*. IEEE, Mar.

2007, pages 34–48. ISBN: 0-7695-2764-7. DOI: 10.1109/CGO.2007.4.

[121] Adam Welc, Suresh Jagannathan, and Antony Hosking. "Safe futures for Java". In: *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA '05*. San Diego, CA, USA: ACM, 2005, pages 439–453. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094845.

[122] Adam Welc, Suresh Jagannathan, and Antony L Hosking. "Transactional Monitors for Concurrent Objects". In: *Proceedings of the European Conference on Object-Oriented Programming - ECOOP '04*. 2004, pages 518–541. ISBN: 3-540-22159-X. DOI: 10.1007/978-3-540-24851-4_24.

[123] Robert P. Wilson et al. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers". In: *ACM SIGPLAN Notices* 29.12 (1994), pages 31–37. ISSN: 03621340. DOI: 10.1145/193209.193217.

[124] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. "Uncovering hidden loop level parallelism in sequential applications". In: *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture - HPCA '08*. Ieee, 2008, pages 290–301. ISBN: 9781424420704. DOI: 10.1109/HPCA.2008.4658647.