Thread-Level Speculation with Kernel Support

Clemens Hammacher Kevin Streit Andreas Zeller Sebastian Hack

Computer Science Departement
Saarland University
Saarbrücken, Germany
lastname@cs.uni-saarland.de

Abstract

Runtime systems for speculative parallelization can be substantially sped up by implementing them with kernel support. We describe a novel implementation of a thread-level speculation (TLS) system using virtual memory to isolate speculative state, implemented in a Linux kernel module. This design choice not only maximizes performance, but also allows to guarantee soundness in the presence of system calls, such as I/O. Its ability to maintain speedups even on programs with frequent mis-speculation, significantly extends its usability, for instance in speculative parallelization. We demonstrate the advantage of kernel-based TLS on a number of programs from the Cilk suite, where this approach is superior to the state of the art in each single case ($7.28\times$ on average). All systems described in this paper are made available as open source.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Thread-level speculation, speculative parallelization, shared memory, kernel module, virtual memory

1. Introduction

With the rise of multi-core processors in off-the-shelf hardware, the need for parallel software has grown in order to utilize the available computation resources. Because parallelizing programs manually is hard and error-prone, several techniques for *automatic parallelization* have been proposed in the past. Extracting parallelism out of general-purpose programs is hard: Their control flow is usually input-dependent and the data structures used can typically not be satisfactorily analyzed by static dependence analyses.

Therefore, an important line of research in automatic parallelization relies on *speculation*. In *speculative parallelization*, the compiler parallelizes code although it cannot prove the correctness of this transformation *statically.* Instead, it relies on a runtime component, the speculation system, to detect conflicts that would violate the sequential semantics. This kind of optimistic parallelization is often called *thread-level speculation* or TLS. Keeping the overhead of the speculation system low is decisive for successful speculative parallelization. Doing so even in the presence of significant mis-speculation gives the compiler the freedom to parallelize code showing statically unpredictable memory access patterns.

Many approaches for speculative parallelization have been proposed (see Section 2) using different techniques for separating speculative state, tracking memory accesses and implementing rollbacks. Some rely on special hardware extensions, others use code instrumentation to buffer speculative memory updates until commit. Also, an increasing number of approaches isolate speculative tasks by forking operating system processes to execute speculative work, and rely on the virtual memory system to efficiently coordinate between speculative and non-speculative memory accesses. This paper contributes to the state of the art of this kind of systems.

Since many proposed parallelization schemes rely on speculation, multiple such TLS systems have been developed in order to support the evaluation of individual parallelization approaches (see also Section 2.2). They are mostly ad hoc implementations featuring a specific parallelization scheme or targeting specific hardware or applications, and are typically not described in detail. Because of that as well as different hardware restrictions and usage patterns, it is hard to compare them against each other or new approaches. Also, typically they are not publicly available to be evaluated or used by others.

This paper not only presents two TLS implementations—one resembling the state-of-the-art and one being novel and more efficient—, but also makes both of them publicly available as a basis for further research[1]. The common parallelization scheme which all the approaches use is based on fork-join based parallelization: At one point in time, several parallel tasks are spawned and execution continues once all of them finished. Streit et al. [30] have shown that this simple scheme not only covers task parallelism, but also DOALL and DO-ACROSS style loop parallelism. With minor extensions also limited forms of DSWP [22] can be included.

After an overview over related work in Section 2, this paper makes the following contributions:

1. We present the design and implementation of two TLS systems (Section 3). Both use the virtual memory system for conflict detection. U-TLS is implemented in user space. K-TLS is, to the best of our knowledge, the first TLS that is entirely implemented in kernel space to enable more efficient memory access tracking and commit.
2. Unlike all TLS systems before, K-TLS provides isolation not only of memory effects but also in the possible presence of system calls like I/O.

CC'16, March 17–18, 2016, Barcelona, Spain
ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892221

[1] All software (STM, U-TLS and K-TLS) is available under the GPL license at `https://github.com/hammacher/k-tls`

3. We evaluate the kernel-based implementation against two state-of-the-art speculation systems: STM and U-TLS (Section 4). First, we measure and compare the overhead of specific TLS operations (spawning, memory tracking, validation and commit). Second, we evaluate the overall system performance on multiple automatically parallelized programs from the Cilk suite of sample applications. Our evaluation demonstrates that implementing a TLS in kernel space leads to significant improvement in all overheads we measure. For large tasks, it can reach the performance of unprotected parallel execution and shows good performance even in the presence of misspeculation.

2. State of the Art

The sheer number of STM and TLS approaches published in the literature demonstrates the need of for speculation systems. This chapter describes their state of the art. Since most recent automatic parallelization approaches target compiled languages like C or C++, we exclude approaches which are implemented in a managed execution environment like the Java virtual machine. We also do not give an overview of parallelization techniques, since this is beyond the scope of this paper.

2.1 Software Transactional Memory

Even before the potential of *thread-level speculation* for automatic parallelization was discovered in the late 1990s (e.g. [19, 29]), a quite similar concept was introduced as *transactional memory* by Herlihy and Moss in 1993 [11]. Two years later, Shavit and Touitou described and implemented the first software-only implementation, since then called *STM* [27]. It already featured word-level conflict detection. In 2006—after the rise of multi-core processors—many new implementations have been proposed, introducing new concepts of implementing STM like global timestamps and lazy snapshotting [5, 24, 26, 34]. Even though designed as an alternative to lock-based parallel programming, STM can also be used to implement TLS [17]. 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. In this work, we compare the performance of the TLS systems we present against a state-of-the-art STM system called TinySTM [7, 8].

2.2 Thread Level Speculation

As with transactional memory, TLS can either be implemented entirely in software, or as a hardware extension. The first work describing speculative parallelization of loops with runtime checks was the LRPD test by Rauchwerger and Padua [23]. Since the main purpose of this work was removing dependences by privatization and reduction recognition, any data dependence which was not statically detected as one of these would cause sequential re-execution. Three years later—in 2002—the first true TLS implementation was presented by Rundberg and Stenström, called S-TLS [25] and written in pure assembly. It uses shadow memory to track read and written memory regions and assign locks to them, and to hold the updated values for a fixed number of parallel tasks. This scheme was later improved by Cintra and Llanos [4] 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 [31] and write-through [18] 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.

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* [9] and others [28, 29]. In 2006, Liu et al. published the POSH compiler [16], 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 [33], a speculative extension of *Decoupled Software Pipelining (DSWP)* [22].

Johnson et al. [13] provide performance measures of their parallelization approach on a simulated *speculative multi-core processor*. This hardware precisely detects true data dependences 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 [12] 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.

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 efficiently in hardware, it can be very efficient.

The first approach using *unix processes to isolate speculative state* is *behavior oriented programming (BOP)* by Ding et al. [6]. 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), and written pages are mapped back to the master process. Even though BOP is not tailored to fork-join parallelization, it could probably be modified for this use-case also. A similar approach by a subset of the authors uses the same technique for *Fast Track* [14], where optimistically optimized code is executed in the main process, while the original code is executed concurrently for validation.

Later, Berger et al. [1] 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. [21] also use separate address spaces to isolate speculative states, but still track memory accesses explicitly and replay them in a central commit unit. Pyla et al. [20] 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. [15] describe a TLS based on memory page protection designed for clusters, with a dedicated validator and commit process.

Even though many approaches make use of virtual address translation in the context of TLS, none of them is publicly available. This paper presents two easy-to-use open source solutions for virtual memory based TLS, one in user space only, and a more efficient variant utilizing a Linux kernel module.

3. TLS Implementations

A speculative runtime system for TLS can be implemented in many different ways, implicating different restrictions on the execution environment and inducing overhead at different points in time. This chapter describes two implementations using virtual memory to isolate speculative states. One of these implementations is novel by using a kernel module for maximum performance and improved soundness guarantees in the presence of I/O. Both implementations execute speculative tasks in separate processes instead of threads. Those are forked from the main process at spawn time, and discarded on a rollback or after committing the memory changes back to the main process.

The interface used by a developer or automatic parallelizer is the same for all implementations: A *task list* is created, containing the tasks to be executed in parallel. Each task consists of a pointer to a function containing the user code, a pointer to memory containing input values for this code, and a pointer to space for the output values. The latter two pointers are passed to the function, but only the second one may be written to. The order of the tasks in the list determines their commit order, hence it should match the sequential order of the tasks. The restriction to a list, implicating the requirement to establish a linear commit order, is a mere technical one. The extension to a commit graph, allowing for parallel commits is left for future work. The task list is passed to the speculation system for parallel execution. This invocation returns not before all tasks have executed and committed successfully.

3.1 Virtual-Memory Based TLS in User Space (U-TLS)

Just like STM and most TLS systems, the U-TLS system works entirely in user space. It communicates with the operating system using system calls that are not wrapped by standard libraries. Therefore, it is usually not portable. A major benefit over STM is that the executed code does not need to be instrumented; calling external functions, e.g., from a pre-compiled library, is allowed within transactions. The remainder of this section describes all the details of the U-TLS implementation, whose pseudo-code implementation is given in Algorithm 1.

3.1.1 Forking Speculative Tasks

In order to execute a task list in U-TLS, the developer or 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 its execution, like the id of the process executing this task (*state.pid*), the read and written pages (*state.read_pages* and *state.modified_pages*), a condition variable to notify the parent of completion of the user code (*state.ready*), and a flag to indicate whether the task actually executing the respective use code completely (*state.finished*). 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 9 to 13, cp. Section 3.1.3). 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 14 to 18).

Table 1. The characteristics of the different speculative runtime systems examined in this paper. STM and U-TLS resemble the state of the art. K-TLS is novel by utilizing a Linux kernel module. U-TLS and K-TLS are described in detail, and their respective trade-offs are discussed.

	pure user space	instrumentation	OS support	efficient setup	efficient execution	efficient commit	fine granular
STM	+	+	-	+	-	-	+
U-TLS	+	-	0	-	+	0	-
K-TLS	-	-	+	-	+	+	-

3.1.2 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 20 to 25). 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. 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 31 to 39), 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 3.1.3). 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 26), 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 30). If available, we use the *vmsplice* system call for this, which has potentially better performance than just writing the data page by page.

3.1.3 Validation 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

Algorithm 1 Pseudo-code implementation of U-TLS

```
 1: procedure UTLSRUN(tasks)                                  ▷ main routine
 2:     $N \leftarrow len(tasks)$
 3:     $ctx \leftarrow allocate\ TLSContext$
 4:     $states \leftarrow allocate\ shared\ TaskState[N]$
 5:     for $i \leftarrow 0$ to $N - 1$ do
 6:         $states[i].pipe \leftarrow$ PIPE( )
 7:         $states[i].pid \leftarrow$ FORK$(runTask, tasks[i], states[i])$
 8:     $comp \leftarrow 0$
 9:     while $comp < N$ do
10:         if COMMIT$(tasks[comp], states[comp], ctx)$ then
11:             $comp \leftarrow comp + 1$
12:         else
13:             break
14:     for $i \leftarrow comp$ to $N - 1$ do
15:         KILL$(states[i].pid)$
16:     for $i \leftarrow comp$ to $N - 1$ do
17:         $functionPtr \leftarrow tasks[i].fun$
18:         $functionPtr(tasks[i].in, tasks[i].out)$

19: procedure RUNTASK(task, state)                            ▷ child process
20:     save state pointer into global variable (process-local)
21:     allocate a new stack and set RSP
22:     protect whole memory (except own stack)
23:     install segmentation fault handler SEGFAULT
24:     $state.read\_pages \leftarrow \varnothing$
25:     $state.modified\_pages \leftarrow \varnothing$
26:     RUNUSERCODE(task)
27:     $state.finished \leftarrow true$
28:     NOTIFY(state.ready)
29:     for each page_addr in state.modified_pages do
30:         WRITE$(state.pipe, page\_addr, PAGE\_SIZE)$

31: procedure SEGFAULT(page_addr)                             ▷ segfault handler
32:     if $page\_addr \in state.modified\_pages$ then
33:         abort
34:     if $page\_addr \in state.read\_pages$ then
35:         add page_addr to state.modified_pages
36:         grant write access to page (with COW)
37:     else
38:         add page_addr to state.read_pages
39:         grant read-only access to page

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

53: procedure VALIDATE(state, ctx)                            ▷ conflict checking
54:     for each page_addr in state.read_pages do
55:         if $page\_addr \in ctx.modified\_pages$ then
56:             return false
57:     return true
```

prematurely (line 41). 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 42) 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 considered failed also.

If any of these two checks fail, the task needs to re-execute (lines 44 to 47). 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 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 18), 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 49 to 51). 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.

3.1.4 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 one 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 to 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* [5, 24]. U-TLS does not contain this optimization for two reasons: First, our benchmarks don't spawn more tasks than hardware threads, which is also ensured by most automatic parallelization systems. Second, U-TLS mostly serves as an evaluation vehicle for K-TLS.

3.1.5 Restrictions of U-TLS

Obviously the U-TLS system can only be used on Unix systems offering the operating system facilities needed, in particular copy-on-write process forking, protecting individual pages for either read-only or no access, customized segmentation fault handlers, and inter-process communication to copy back changed data.

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. Unwanted 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.

3.2 Virtual-Memory Based TLS In Kernel Space (K-TLS)

K-TLS works similar to U-TLS, but the major operations are executed in the operating system kernel, instead of from user space. This not only improves efficiency, but also allows to handle I/O and other system calls from within speculative tasks, providing general protection of arbitrary user code (see Section 3.2.5). This section describes the design of the kernel code as shown in Algorithm 2.

3.2.1 User-Space Interface

The interface for starting speculative parallel execution in K-TLS is identical to the one of U-TLS (cp. Section 3). The big difference is that the task list is not processed, but just passed 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).

3.2.2 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. The *KTLSContext* consists of a pointer to the kernel structure for the parent process (*parent_task*), and a map which stores the *version number* of each page modified by any speculative task. This version number is the sequential number of the last task which modified this page, and is used for conflict detection (see Section 3.2.4).

Then, the individual tasks are forked as described in the next section. Afterwards, they are committed in order (lines 19 to 28). 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 3.2.4). If this validation fails, the task is re-spawned with an up-to-date view of all memory changes committed so far (lines 23 to 24). 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) does complete the execution of the user code (line 25). If this check fails, this 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.

3.2.3 Execution of Speculative Tasks

After a new child process is forked from the main process (line 33), its memory is made inaccessible by clearing all pages of the respecting *virtual memory area (VMA)* from the page table, and registering our own page fault handler for this VMA. 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 41 to 45). 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 (cp. Section 3.2.5) happen. Page faults are handled in the PAGEFAULT routine. 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 sequence number of the last committed task (*task.ctx.version*). This ensures that there are no false conflicts reported if another task commits between the fork point and the first memory access of the current task. Then, the page is looked up in the page table of the parent task (line 57). 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.

3.2.4 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 64). 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 task, the kernel module compares the physical pages this address maps to in the task and the parent process (lines 72 to 73). If those pages differ, then 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. Afterwards, we still need to copy the direct output of the task (cp. Section 3) from the child's stack back to the parent (lines 77 to 78).

3.2.5 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[2] 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.

3.2.6 Optimizations

For the sake of simpler presentation, we slightly simplified some of the implementation details in the previous sections. For performance reasons, the actual implementation sometimes deviates from

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

Algorithm 2 Pseudo-code implementation of K-TLS

```
 1: procedure RUNTASKS(tasks)                          ▷ user-space interface
 2:     $fd \leftarrow$ OPEN("/dev/ktls")
 3:     $numExec \leftarrow$ IOCTL($fd$, $KTLS\_RUN$, $tasks$)
 4:     for $i \leftarrow numExec$ to $len(tasks) - 1$ do
 5:         $functionPtr \leftarrow tasks[i].fun$
 6:         $functionPtr(tasks[i].in, tasks[i].out)$

 7: procedure KTLSRUN(tasks)                           ▷ kernel-space entry
 8:     $N \leftarrow len(tasks)$
 9:     $ctx \leftarrow$ allocate KTLSContext
10:     $ctx.parent\_task \leftarrow current$
11:     $ctx.start\_version \leftarrow 0$
12:     $ctx.page\_versions \leftarrow$ allocate hash map
13:     $states \leftarrow$ allocate TaskState$[N]$
14:     for $i \leftarrow 0$ to $N - 1$ do
15:         $states[i].sequence\_nr \leftarrow i$
16:         $states[i].ctx \leftarrow ctx$
17:         SPAWNTASK($tasks[i]$, $states[i]$)
18:     $exec \leftarrow 0$
19:     while $exec < N$ do
20:         WAITFORCOMPLETION($states[exec].task$)
21:         $valid \leftarrow$ VALIDATE($ctx$, $states[exec]$)
22:         if not $valid$ then
23:             SPAWNTASK($tasks[exec]$, $states[exec]$)
24:             WAITFORCOMPLETION($states[exec].task$)
25:         if $states[exec].task.exit\_code \neq 0$ then
26:             break
27:         COMMIT($ctx$, $states[exec]$)
28:         $exec \leftarrow exec + 1$
29:     for $i \leftarrow exec$ to $N - 1$ do
30:         KILL($states[i].task$)
31:     return $exec$

32: procedure SPAWNTASK(task, state)                   ▷ spawn new task
33:     $state.task \leftarrow$ COPYPROCESS($current$)
34:     PROTECTMEMORY($state.task$)
35:     $state.touched\_pages \leftarrow$ allocate hash set
36:     $state.stack \leftarrow$ ALLOCATEVMA($state.task$, $16 * 2^{20}$)
37:     $stackTop \leftarrow state.stack + 16 * 2^{20}$
38:     $outputSpace \leftarrow stackTop - len(task.output)$
39:     $inputSpace \leftarrow outputSpace - len(task.input)$
40:     $inputSpace[0 : len(task.input)] \leftarrow task.input$
41:     $state.task.regs.rbp \leftarrow inputSpace$
42:     $state.task.regs.rsp \leftarrow inputSpace$
43:     $state.task.regs.rip \leftarrow task.fun$
44:     $state.task.regs.rdi \leftarrow inputSpace$
45:     $state.task.regs.rsi \leftarrow outputSpace$
46:     SCHEDULETASK($state.task$)

47: procedure PROTECTMEMORY(task)                      ▷ setup virt. memory
48:     for each $vma$ in $task.vmas$ do
49:         if $vma.flags$ & $VM\_WRITE$ then
50:             CLEARPAGES($task$, $vma$)
51:             $vma.page\_fault\_handler \leftarrow$ PAGEFAULT
52:             $vma.vm\_private\_data \leftarrow task$

53: procedure PAGEFAULT(addr)                          ▷ page fault handler
54:     $state \leftarrow$ FINDVMA($current$, $addr$)$.vm\_private\_data$
55:     if $state.touched\_pages$ is empty then
56:         $state.start\_version \leftarrow state.ctx.version$
57:     $page \leftarrow$ PAGETABLEWALK($task.parent$, $addr$)
58:     if $page$ exists then
59:         add $addr$ to $state.touched\_pages$
60:         return $page$
61:     else
62:         return $NO\_PAGE$

63: procedure VALIDATE(ctx, state)                     ▷ pre-commit validation
64:     if $state.start\_version = ctx.version$ then
65:         return true
66:     for each $addr$ in $state.touched\_pages$ do
67:         if $ctx.page\_versions[addr] > state.start\_version$ then
68:             return false
69:     return true

70: procedure COMMIT(ctx, state)                       ▷ commit speculative state
71:     for each addr in $state.touched\_pages$ do
72:         $newP \leftarrow$ PAGETABLEWALK($state.task$, $addr$)
73:         $oldP \leftarrow$ PAGETABLEWALK($ctx.parent\_task$, $addr$)
74:         if $newP \neq oldP$ then
75:             PAGETABLEUPDATE($ctx.parent\_task$, $addr$, $newP$)
76:             $ctx.page\_versions[addr] \leftarrow state.sequence\_nr$
77:     $outputSpace \leftarrow state.stack + 16 * 2^{20} - len(task.output)$
78:     $task.output \leftarrow outputSpace[0 : len(task.output)]$
```

the description in the text. We give an overview over these optimizations below.

In Algorithm 2, 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 stack VMA and the instruction pointer, and schedules the task.

We also optimize the actual forking of the task: 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 50), we just skip copying the respective VMAs and associated page table entries, and allocate new VMAs during the aforementioned setup. 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 4.1.1), 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.

4. 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. Additionally, several non-K-TLS benchmarks are run directly on the host system. We verify that the time measures match those in the virtual machine. The host system is equipped with a quad-core Intel i7 870 CPU running at 2.93 GHz and 16 GB of main memory. The virtual machine has access to all four CPUs, and 8 GB of memory. For each benchmark, we report the arithmetic mean over 10 runs. No other processes were executing on both the host and the guest system.

We compare the U-TLS and K-TLS systems described in Section 3 against the state-of-the-art STM system TinySTM, which provides similar guarantees as TLS. We improved the internal us-

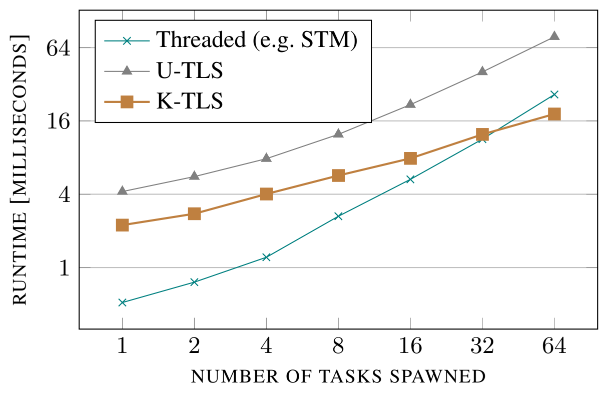


Figure 1. Overhead of spawning a task list of varying size. Since STM uses threads instead of processes, it spawns tasks the fastest (around 0.3 ms). U-TLS uses a *fork* system call per task, which takes around 1.2 ms, plus initial 4 ms per task list. K-TLS forks the process directly in the kernel, requiring less context switches for protecting memory. K-TLS takes around 1.5 ms initially plus 0.6 ms per spawned task. Additionally, it reuses tasks once they finish execution, leading to less actually forked tasks for larger numbers.

age of data structures of TinySTM to achieve constant complexity per access, instead of linear in the original implementation. A similar approach was proposed by Harris et al. [10]. This is necessary because the memory footprints of TLS tasks are typically much larger than those of traditional transactional memory benchmarks. Also, automatic parallelizers typically exclude tasks which are too small for the parallelization and communication overhead to pay off. In order to preserve the sequential semantics during speculative parallel execution in STM, we establish a linear *commit order* by waiting for all predecessor tasks to finish before starting the commit phase.

4.1 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. For each of these figures, we run a microbenchmark as described in the following sections.

4.1.1 Spawning Tasks

Most runtime systems that track memory accesses explicitly (like STM) execute parallel tasks in individual threads. U-TLS and K-TLS use virtual memory and therefore have to fork processes which execute the tasks. Hence, their initial overhead is larger, whereas the overhead during task execution is smaller.

In this benchmark, each run creates a task list of N empty tasks, where N varies between 1 and 64. We measured the overall wall-clock execution time of executing this task list for the different systems (STM, U-TLS, K-TLS). 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. All 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 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 or page table. 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. U-TLS takes about 4 ms for spawning a single task via the *fork* system call, 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. K-TLS takes about 2 ms for spawning the first task, and between 0.5 and 0.6 ms per additional task. These numbers are lower than for U-TLS because creating new processes and protecting its memory requires less context switches if executed directly in the kernel (cp. Section 3.2.6). Additionally, K-TLS reuses the processes after finishing the execution of one task. This reduces the cost per additional task to below 0.2 ms.

4.1.2 Execution Overhead

The second source of overhead is the tracking of 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. In U-TLS and K-TLS the overhead is caused by two actions: copying memory pages which have been written, and context switches between user space and kernel space for each page fault. K-TLS reduces context switches by handling the respective page faults directly in the kernel.

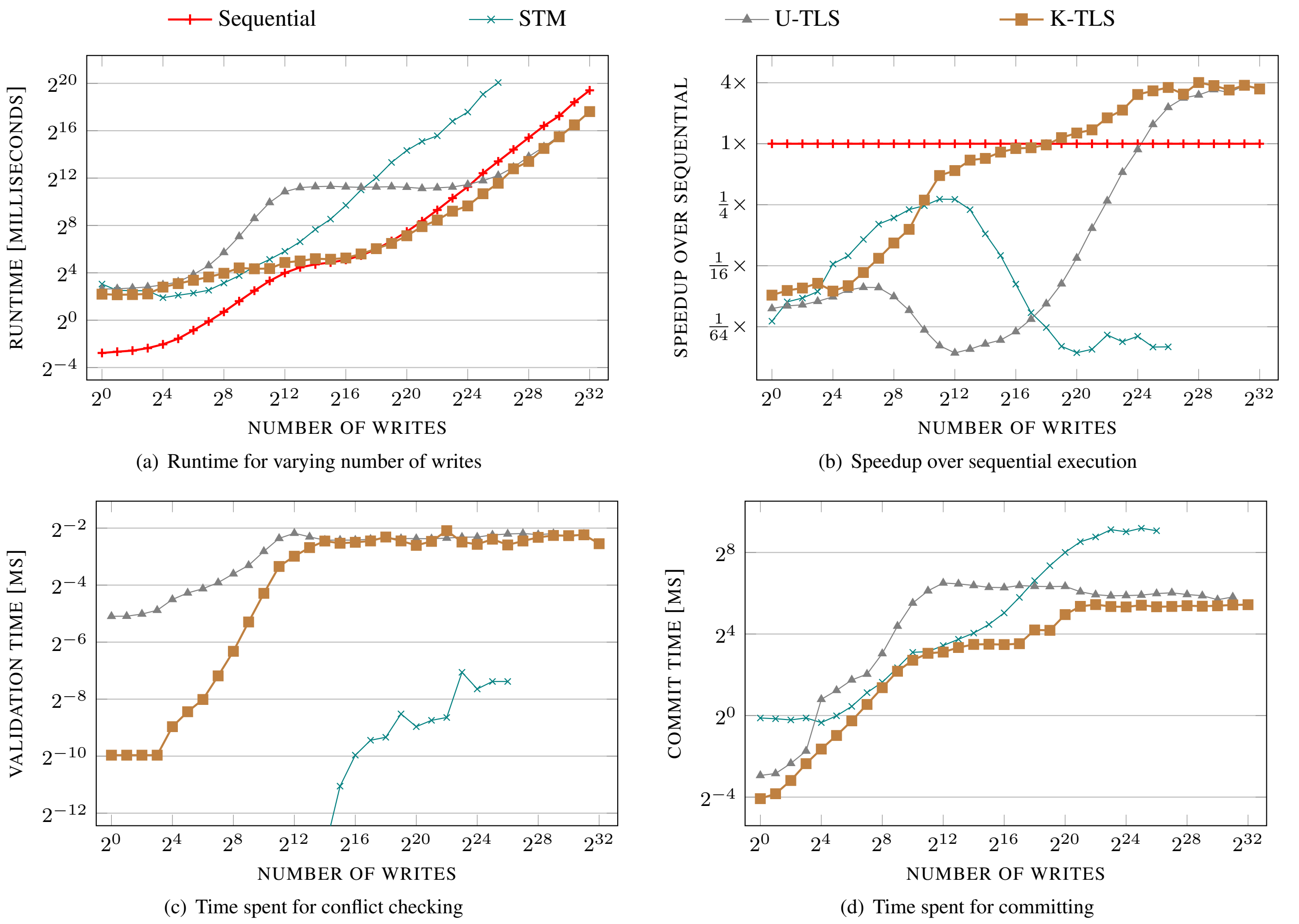
In order to compare the runtime overhead of the different systems, we run a benchmark in which four parallel tasks perform random write accesses to disjoint memory blocks. The memory area which is updated by each task has a size of 16 MB. Figure 2(a) shows the runtime for every system with respect to a varying number of memory accesses. Figure 2(b) shows the corresponding speedup against sequential execution. Note that all numbers are reported on a logarithmic scale on both axes.

For a very low number of memory accesses (up to 128), the most efficient execution is sequential. Above this limit, parallelization without any runtime system gives a speedup. Interestingly, between 2^{12} and 2^{17}, the speedup is even super-linear, probably due to better cache utilization in the parallel cores. STM and U-TLS never succeed to beat the sequential runtime. For STM, part of this bad performance can be explained by the rollbacks it performs. Because TinySTM maps memory addresses to a *lock array* of fixed size, there are hash collisions which provoke false rollbacks. From 2^{16} on, TinySTM executes, on average, more than one rollback per execution of four tasks, and reaches a 50 % rollback rate for more writes. U-TLS has surprisingly high overheads for mid-sized number of writes. Profiling this 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 VMAs up to several thousands. The Linux kernel organizes the virtual memory descriptor as a linked list. Because it is traversed on every page fault (and other operations), it 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. K-TLS on the other hand outperforms sequential execution from 2^{18} accesses upwards, and converges to the *unsafe* parallel execution at higher numbers. It avoids the problem of fragmenting the virtual memory descriptor because it directly updates page table entries instead of virtual memory areas. This is only possible in kernel space.

4.1.3 Validation and Commit

After parallel execution, all systems have to validate the set of reads and writes (commit). STM does this on the granularity of a word in software, U-TLS and K-TLS use page level granularity for both tasks. Figures 2(c) and 2(d) show the time spent in validation and commit.

The validation time is negligible for all configurations. For the missing STM values, we measured less than one microsecond per



(a) Runtime for varying number of writes

(b) Speedup over sequential execution

(c) Time spent for conflict checking

(d) Time spent for committing

Figure 2. Performance of the different runtime systems on 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. K-TLS consistently outperforms U-TLS. It provides speedups over sequential execution for mid-sized to large-sized matrices, and finally reaches the perfect speedup of 4X.

run. Validation time never exceeds one percent of overall execution time, for most configurations it is orders of magnitude lower.

The time spent for committing is much larger. For both TLS systems 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. From this point on, the commit time stabilizes because no new pages have been touched. STM commits faster than U-TLS in the midrange, where memory accesses are very sparse on the pages. Then, it increases further, until also most individual words have been written at least once. K-TLS shows the least commit time over the full range.

4.2 Usage in Automatic Parallelization

Section 4.1 shows that all operations of a TLS system are sped up by implementing them directly in the operating system. This section evaluates how these performance benefits translate into more speedup of speculatively parallelized programs.

To this end, we integrated all three systems (STM, U-TLS and K-TLS) into the automatic parallelization framework *Sambamba* [30]. The compiler extracts functions for the individual tasks, passes input and output values via an individual memory structure, and places calls to the API of TLS to create a new list, add a task to the list, spawn the execution of the list, and finally delete the list. If a loop is to be parallelized, the system allocates a task list before entering the loop, and in each iteration adds the corresponding task(s) to the list. Once 16 tasks are collected (or the loop exits), the task list is executed and then cleared.

The benchmarks we have chosen for this evaluation are the *serial elision* of the *Cilk* [2] program suite. 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 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.

Table 2 lists the programs used for the evaluation, and their respective parameters. 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 shortly describes the programs which could be speculatively parallelized.

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

Table 2. Characteristics and performance of eight programs from the Cilk program suite, automatically parallelized using the *Sambamba* framework [30]. 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. 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. We do not report numbers for STM, because it did not terminate within two hours on six of the programs, and when it terminates, it performs much worse than U-TLS.

program	input size	number of speculative tasks	number of rollbacks	percentage of rollbacks	avg. number of read pages	avg. number of written pages	avg. runtime per task	overall runtime sequential	overall runtime K-TLS	overall runtime U-TLS	speedup K-TLS	speedup U-TLS
Cilksort	4194304	8	0	0 %	6148	4096	347.3 ms	2.06 s	0.99 s	3.32 s	2.09×	$\frac{1}{1.61}\times$
Fft	4194304	18	1	5.6 %	7769	1294	102.5 ms	1.21 s	1.04 s	126.21 s	1.17×	$\frac{1}{104}\times$
Heat	4096x1024x100	204	2	1.0 %	8088	4056	47.9 ms	3.72 s	6.88 s	2506.25 s	$\frac{1}{1.85}\times$	$\frac{1}{674}\times$
Lu	2048	756	147	19.44 %	49	18	16.9 ms	13.44 s	6.19 s	9.62 s	2.17×	1.40×
Matmul	2048	2	0	0 %	8195	2048	25764.2 ms	49.64 s	24.88 s	39.66 s	1.99×	1.25×
Plu	2048	150	0	0 %	327	135	99.9 ms	12.95 s	6.12 s	9.21 s	2.12×	1.41×
Spacemul	2048	8	0	0 %	6149	2048	2814.3 ms	17.87 s	5.14 s	5.45 s	3.48×	3.28×
Strassen	2048	8	0	0 %	6149	2048	3467.8 ms	21.90 s	7.55 s	100.98 s	2.90×	$\frac{1}{4.61}\times$
										Geometric mean:	1.82×	$\frac{1}{3.99}\times$

Fft an implementation of fast fourier transform.

Heat which 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 applied the parallelization analysis of *Sambamba*, and placed manual parallelization hints for speculation because *Sambamba* does not support speculation yet. In all programs, we changed the memory allocation such that large objects are automatically aligned to 4096 bytes. 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 all memory allocation sites.

The parallelization is straightforward to apply in all cases. Between one and four locations are parallelized in each program. 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 re-implement these algorithms without the need for speculation. Automatic approaches however can not.

Figure 3 plots the speedups measured on the eight programs, with error bars showing the standard deviation. We observe that while U-TLS is only able to speed up four of the programs, K-TLS provides speedups for seven of them. Also, K-TLS is superior to U-TLS in all cases. The geometric mean of the speedup of K-TLS is $1.82\times$, while U-TLS shows a slowdown of $3.99\times$. This sums up to a $7.28\times$ speedup of K-TLS over U-TLS. We also ran the programs with STM instrumentation, but—like others before [1, 3, 32]—observe that it is unusable in the context of automatic parallelization. STM only terminated on two programs within a timeout of two hours. On *Cilksort* it shows slowdowns of $13\times$, and on *Plu* of $78\times$, which is much slower than U-TLS.

Interestingly, both U-TLS and K-TLS still provide speedup for the *Lu* program, where the percentage of rollbacks is close to 20 %. These slowdowns 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 TLS even provides speedups for low to medium sized parallel tasks.

The large slowdowns of U-TLS are often caused by scattered memory accesses, which partition the virtual memory area of the process (see Section 4.1.2). On *Fft* and *Strassen*, this is caused by the accesses to the shared array. *Heat* allocates the rows of the grid in the heap, so they are not consecutive either. *Heat* is also the only program which executes system calls inside of speculative

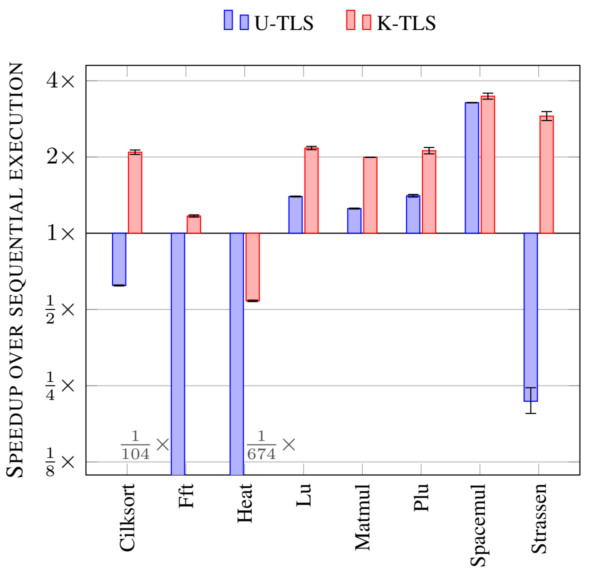


Figure 3. Comparison of the speedup achieved by automatic parallelization of eight programs from the *Cilk* suite. K-TLS outperforms the state-of-the-art U-TLS as well as STM (not shown in this plot because all bars are far below the negative limit).

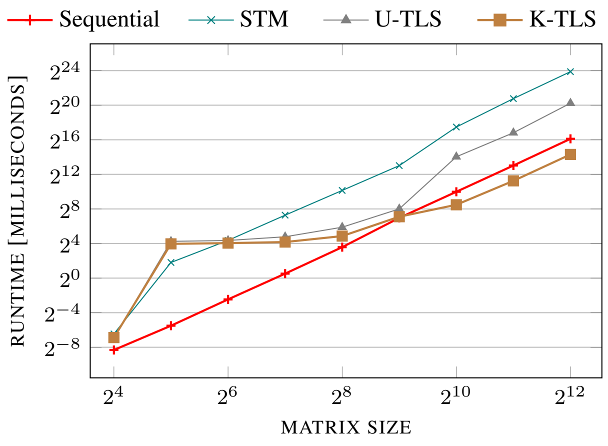
tasks (in this case a *brk* system call to allocate more heap space). It therefore triggers a rollback in K-TLS. In U-TLS, the respective tasks were manually excluded because the parallel program would crash otherwise.

4.2.1 Performance vs. Input Size

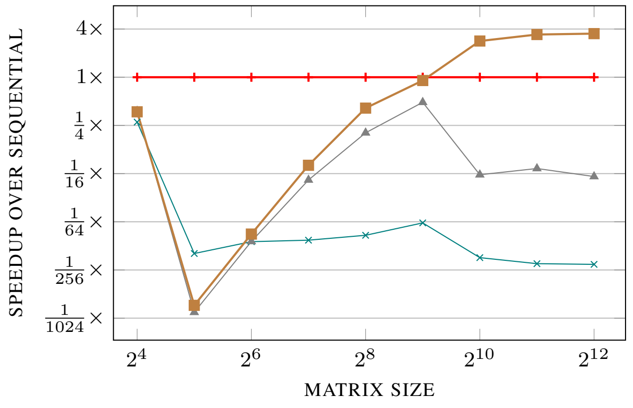
To further evaluate the performance of speculatively parallelized programs on varying input sizes, we chose the *Strassen* program for further investigation. Remember that *Strassen* recursively processes quarters of the matrix, hence the memory accesses are not consecutive, but in a striped manner. This also makes it particularly difficult to prove the memory accesses in the parallel tasks disjoint, hence speculation is required for automatic parallelization.

Figures 4(a) and 4(b) show the result of executing this program either sequentially, or parallel with STM, U-TLS or K-TLS for conflict detection. The overall picture is comparable to Figure 2, but this time U-TLS does not show huge overheads. This is because the Strassen algorithm writes the matrix cells successively, so the virtual address space does not as fragmented as for random accesses. At the smallest measured size of 16, the implementation does not recurse, so no parallelization is performed. The parallel code path exists though, resulting in less optimized code being generated also for the executed sequential code path. This causes the slowdown of several microseconds.

For matrices up to 512×512 elements, each row spans not more than one memory page, so the quarters which are processed in parallel share memory pages. Therefore, both K-TLS and U-TLS detect memory conflicts, and need to re-execute each task. For matrices above this limit, K-TLS again converges to the parallel execution without a runtime system. All other systems fail to give speedups in this benchmark. For U-TLS, we observe a performance loss from 1024 upwards. This is because for smaller matrices the modified pages form continuous regions, and these are much more efficient to communicate through the pipe (cp. Section 3.1.2) than the individual pages for larger sizes.



(a) Runtime of the strassen algorithm



(b) Speedup over sequential execution

Figure 4. Performance of the automatically parallelized *Strassen* implementation from the *Cilk* program suite. Both STM and U-TLS fail to provide any speedup. K-TLS outperforms sequential execution once the matrices are large enough such that each row spans multiple memory pages. In this case, we come close to the perfect speedup of $4\times$.

5. Conclusion

We have presented K-TLS that is, to the best of our knowledge, the first thread-level speculation system that uses the virtual memory system and is implemented entirely in kernel space. Unlike existing TLS systems, K-TLS not only provides isolation of memory effects but also for I/O effects and other system interactions. Our experimental evaluation shows that K-TLS outperforms user-space TLS systems in terms of spawning tasks, conflict detection, and commit. Furthermore K-TLS is able to maintain performance even in the presence of a non-negligible amount of rollbacks. Where the overhead of the user-space system amortizes only for speculative tasks with a very large footprint, K-TLS converges much earlier against the performance of unprotected parallel execution. For a speculatively parallelized version of the Strassen matrix multiplication algorithm, K-TLS is the only speculation system able to match the speedup of unprotected parallel execution for medium input sizes. Our results indicate that kernel-space TLS systems can provide significant efficiency gains as well as improved soundness guarantees for speculative parallelization.

References

[1] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA '09*, pages 81–96, 2009.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95*, pages 207–216, 1995.

[3] C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, nov 2008.

[4] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03*, pages 13–24, 2003.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC '06*, pages 194–208, 2006.

[6] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07*, pages 223–234, 2007.

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08*, pages 237–245, 2008.

[8] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-Based Software Transactional Memory. *IEEE TPDS*, 21(12):1793–1807, Dec. 2010.

[9] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *Micro*, 20(2):71–84, 2000.

[10] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *ACM SIGPLAN Notices*, 41(6):14–25, June 2006.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, pages 289–300, 1993.

[12] B. Hertzberg and K. Olukotun. Runtime automatic speculative parallelization. In *CGO '11*, pages 64–73, 2011.

[13] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07*, pages 205–214, 2007.

[14] K. Kelsey, C. Zhang, and C. Ding. Fast Track: Supporting Unsafe Optimizations with Software Speculation. In *PACT '07*, pages 414–429, Sept. 2007.

[15] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative DOALL for clusters. In *CGO '12*, pages 94–103, 2012.

[16] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *PPoPP '06*, pages 158–167, Mar. 2006.

[17] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09*, pages 166–176, 2009.

[18] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation.

[19] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical report, 1997.

[20] H. K. Pyla, C. Ribbens, and S. Varadarajan. Exploiting coarse-grain speculative parallelism. In *OOPSLA '11*, pages 555–574, 2011.

[21] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS '10*, pages 65–76, 2010.

[22] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT '04*, pages 177–188, 2004.

[23] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE TPDS*, 10(2):160–180, 1999.

[24] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In S. Dolev, editor, *DISC '06*, pages 284–298, 2006.

[25] P. Rundberg and P. Stenström. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *Journal of Instruction-Level Parallelism*, 3(2001), 2002.

[26] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06*, pages 187–197, Mar. 2006.

[27] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*, pages 204–213, 1995.

[28] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *ACM SIGARCH Computer Architecture News*, 23(2):414–425, May 1995.

[29] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98*, pages 2–13, 1998.

[30] K. Streit, J. Doerfert, C. Hammacher, A. Zeller, and S. Hack. Generalized Task Parallelism. *TACO*, 12(1), 2015.

[31] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Micro '08*, pages 330–341, Nov. 2008.

[32] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI '10*, pages 62–73, June 2010.

[33] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07*, pages 49–59, 2007.

[34] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *CGO '07*, pages 34–48, Mar. 2007.