

An Analysis of I/O And Syscalls In Critical Sections And Their Implications For Transactional Memory

Lee Baugh and Craig Zilles

Dept. of Computer Science, University of Illinois at Urbana-Champaign
[leebaugh, zilles]@cs.uiuc.edu

Abstract

Transactional memory (TM) is a scalable and concurrent way to build atomic sections. One aspect of TM that remains unclear is how side-effecting operations – that is, those which cannot be transparently undone by a TM system – should be handled. This uncertainty poses a significant barrier to the general applicability and acceptance of TM. Further, the absence of transactional workloads makes it difficult to study this aspect

In this paper, we characterize the usage of I/O, and in particular system calls, within critical sections in two large applications, exploring both the actions performed and the characteristics of the critical sections in which they are performed. Shared memory programs employing critical sections are the closest approximation available to transactional workloads, so using this characterization, we attempt to reason about how the behavior we observed relates to the previous proposals for handling side-effecting operations within transactions. We find that the large majority of syscalls performed within critical sections can be handled with a range of existing techniques in a way transparent to the application developer. We also find that while side-effecting critical sections are rare, they tend to be quite long-lasting, and that many of these critical sections perform their first syscall (and thus become side-effecting) relatively early in their execution. Finally, we show that while these long-lived, side-effecting critical sections tend to execute concurrently with many critical sections on other threads, we observe little concurrency between side-effecting critical sections.

1 Introduction

Transactional Memory (TM) is a language- and system-level technique for concurrent programming. Transactions – atomic regions optimistically executed concurrently – are generally thought to be easier to program than locks, reducing exposure to deadlock, permitting easier composition of code regions, and simplifying atomic semantics. For these and other reasons, there are numerous recent proposals for both software- and

hardware-based TM systems. [2, 6, 8–11, 14, 18, 19]

TM, however, only provides clean semantics for speculatively accessing cacheable shared-memory storage locations, not side-effects such as I/O. Side-effects, simply defined as effects on logical system state which the TM system provides neither conflict detection nor rollback, are difficult to handle speculatively because they can violate isolation and atomicity. For example, when a transaction which has written to a file is aborted, the file has still been written to.

Interesting programs necessarily have side-effects, such as I/O and system calls. The interaction of these side-effects and TM must be considered if a code region containing side-effects requires isolated access to program state. In our analysis of two large multi-threaded programs, we have found that it is not uncommon for side-effecting actions to be contained in (lock-based) critical sections, and, at least for the bulk of the critical sections we analyzed by hand, the lock was providing isolated access to data, not just ensuring mutual exclusion.

To provide isolated access to a data structure in the presence of side-effecting operations, we have two choices: 1) we can retain the use of locks to protect any data that are accessed by a critical section containing a side-effect; these locks would need to be acquired by any code that accessed those data, even strongly-atomic transactional code, or 2) we can enable TM atomic regions to include side-effects, at least for the commonly occurring cases. The second approach has the potential to provide an overall simpler approach to ensuring isolated data access in the presence of side-effects.

While previous work has proposed a number of mechanisms for handling I/O [3, 6, 8, 13, 15, 16, 20], the paucity of significant TM workloads prevents us from evaluating these proposals in any meaningful way. The goal of this paper is to try to understand how TM programmers will use side-effecting operations and, therefore, how support for them should be architected in TM systems. To shed light on this subject – in the absence of compelling TM workloads – we characterize the use of side-effecting operations, specifically syscalls, in two existing, conventionally synchronized, multithreaded workloads. In particular, we characterize and classify the kinds of side-effecting operations seen in our workloads, what minimum protection they require, and the characteristics of the critical sections that

contain them.

We first provide a view of transactional actions drawn from transactional databases (Section 2), and present a survey of proposed techniques for handling side-effects in transactions (Section 3). Then, on the supposition that contemporary critical sections will provide a lower bound (in size and composition) to the transactions of the future, we examine two large multithreaded workloads – MySQL and Firefox – to characterize the syscalls occurring in their critical sections. We find that most of the syscalls can be encapsulated so that programmers need not be aware of them. Finally, we characterize the critical sections invoking these syscalls themselves, describing some of their structure and examining the impact of syscalling transactions upon the whole program, with a view towards the amount of concurrency that may be lost by forcing syscalling critical sections to execute serially. (Section 5).

2 Types of Actions

Transactional memory has strong roots in the earlier research in transactional databases, which share many properties with TM. When considering how TM might handle side-effecting actions in atomic regions, it is useful to consider a taxonomy for transactional actions presented by Gray and Reuter [5]:

Protected actions are those which can be completely compensated for by the transactional memory system. Providing both failure atomicity and isolation, these are operations which affect only CPU state and memory.

Unprotected actions are those for which the TM system cannot compensate, but for which the programmer may provide compensation code. With correct compensation code, these actions provide atomicity and may, though do not necessarily, provide isolation. Filesystem operations may be considered *unprotected*. It is important to note that the selection of adequate compensation code, and indeed what constitutes such code, is left to the discretion of the programmer.

Real actions are those for which there is no adequate compensation. Gray and Reuter’s canonical example is “launch missile,” but more prosaically, printing a document, deleting a file, or even sending a message across a network may be considered *real* actions. Typically, *real* actions may only be executed when a transaction is known to be nonspeculative. If, however, the programmer is willing to ignore a spuriously printed document, or creates a transaction-safe network protocol, even these actions may be considered merely *unprotected*. In a real sense, what makes an action *real* or *unprotected* is merely what the programmer is willing to tolerate.

Drawing from this classification, we will use the term *protection* to refer to something enabling *real* or *unprotected* actions to be executed from within transactions: a transition to nonspeculative execution, a deferral of *real* or *unprotected* actions

until the transaction is known to be nonspeculative, or the compensation block registered by the programmer on some piece of transactional code, to be executed if the transaction aborts. We call any code which consists entirely of either *protected* actions or *unprotected* or *real* actions with protection *transaction-safe*.

3 Proposals for handling Side-Effects in Transactions

We summarize several proposals which address *unprotected* or *real* actions in transactions:

- **Outlaw:** The most restrictive approach, used in STMs [8], simply forbids any non-*protected* actions from occurring transactionally. While this approach offers simplicity by sidestepping the issue entirely, the limitations on programmability and composition that it places are probably unacceptable for general use.
- **Defer:** In some cases, it may be possible to defer *unprotected* or *real* actions until the transaction is certain to commit [6, 7] or placing them in completion actions [20]. This may be done explicitly by the programmer, as in two-phase commit [13] or may be performed implicitly by the compiler or TM system. While this is suitable for some write-only actions or flushes, it is not a general solution because it prohibits dependences upon the return values of *unprotected* or *real* actions within a transaction, such as checking for an error status. Moreover, programmers expect their code to execute in program order; an automatic reordering could lead to unexpected effects.
- **“Go Nonspeculative”:** Another approach is simply to force transactions about to perform *unprotected* or *real* actions to “go nonspeculative” by acquiring a global commit token [3, 6]. While this technique can accommodate even the most irreversible of *real* actions handily, it does have some drawbacks. Under this regime, an *unprotected* or *real* operation inside of a transaction amounts to an implicit promise, by the programmer, that the transaction will complete. The TM system can be made to guarantee that it will not abort this transaction, but this can limit concurrency by serializing side-effecting transactions, negatively affecting performance. Furthermore, the transaction must not have an explicit abort operation along any execution path after the first *unprotected* or *real* action. Because the promise to complete was implicit, and the *unprotected* or *real* action may have been called deep within library code, programmers may be left uncertain of where explicit abort operations may be used. This in turn affects the application of language-level techniques like `retry` and `orElse` [8].

- **Compensate:** The last, and most complex, approach is to permit the programmer to protect *unprotected* code by associating a compensation block with the *unprotected* code [7, 13, 15, 16, 20], not unlike the use of `catch` blocks to guard possibly-exceptioning code. This approach permits programmers to decide what appropriate compensation is for their *unprotected* actions, allows transactions with *unprotected* actions to execute concurrently, and also permits explicit aborts. However, it also introduces a new source for bugs, and does not implicitly provide isolation or conflict detection on *unprotected* code.

As we will see in Section 4, no one of these techniques clearly subsumes the others with respect to supporting all side-effects in all possible transaction code, or even in the workloads we studied. Outlawing side-effects in transactions is problematic, as discussed in Section 1. If it is not feasible to prohibit all side-effects in atomic regions, then atomic side-effects must be performed in lock-based critical sections. If, however, there is any intersection between the data accessed in transactions and the data accessed in critical sections, then all such transactions will also have to acquire the relevant locks. Deferral can only be used when no return values from side-effecting operations are computed upon later in the transaction. “Going nonspeculative” precludes explicit aborts (and raises difficulties with combining locks and transactions.) Compensation cannot be applied whenever the programmer cannot provide an adequate compensation block. Indeed, all are necessarily incomplete solutions, for some transactional code is simply untenable – consider the case of a transaction which performs a *real* action and later issues an explicit abort.

In this paper, we are concerned with the relative suitability of these approaches. We have the following questions:

- How common are side-effecting operations in critical sections? In other words, how much code would simply outlawing side-effects in transactions affect?
- What kinds of side-effecting operations are seen in critical sections? How much of it must be considered *real* actions, and how much *unprotected*?
- How often are side-effecting operations performed at the end of critical sections? Side-effecting operations performed at the end are less likely to return a value that the transaction operates on, so they are more likely to be deferrable.
- If “going nonspeculative” is required, how early the transaction must do so affects how much concurrency the transaction will permit. How are side-effecting operations distributed throughout the lifetime of side-effecting critical sections, and how long are these critical sections? To what degree do side-effecting critical sections overlap with other critical sections?

- What kinds of compensation do the side-effecting operations in our workloads’ critical sections require?

4 Side-Effects in Critical Sections

For our analysis, we sought out large, complex, multithreaded programs in which I/O and other side-effects is reasonably expected. After exploring a number of programs, we selected two workloads that had non-trivial amounts of side-effecting operations called from within critical sections: MySQL and Firefox. MySQL is a multithreaded SQL database server; our test installation of MySQL uses the InnoDB storage engine. MySQL has a main connection thread and creates a new thread for each client connection it receives, and InnoDB was configured to permit up to 50 concurrent I/O threads. Our profiled runs used SysBench [1] first to prepare (create and populate) a small database, and then to access it. Firefox is a popular web browser. Our profiled runs started Firefox up in a pre-set profile, loaded a series of web pages, then shut it down.

4.1 I/Os and Syscalls

Having selected our workloads, we must specify what side-effects we seek. In Section 1, we defined side-effects as operations which affect system state in a way for which the TM system cannot automatically extend isolation and atomicity. TM systems guarantee isolation and atomicity to CPU state and memory, so in non-transactional code like our workloads, side-effecting operations are device I/O.¹

In x86 code, there are three ways in which I/O may be performed:

- The `in` and `out` instructions permit communication directly with a port, as configured by `ioperm()` or `iopl()`.
- Some system calls can delegate to the kernel to perform I/O work.
- Memory-mapped I/O permits the mapping of memory spaces to devices or files.

The `in` and `out` instructions in application code are quite uncommon in both MySQL and Firefox, and are not seen at all inside of critical sections. Memory-mapped I/O generally happens in kernel code, which (for reasons we shall address shortly) should not be transactionally executed in user-level transactions. One exception to this is X11, which among our workloads only applies to Firefox. However, as our Firefox delegates all its X11 rendering to a single thread we do

¹Several techniques have been proposed which could make side-effects of changes to CPU state or memory, by enclosing these changes in paused regions or open nests [15, 16, 20].

not consider memory-mapped I/O. System calls, on the other hand, are plentiful across all threads, as well as within critical sections – but not all actually perform I/O. All, however, execute kernel code.

Current software transactional memory systems (STMs) *cannot* execute kernel code transactionally, and so in STMs all syscalls are side-effecting, whether or not they actually generate I/O. Nor is it clear that this inability is a weakness of STMs. Zilles and Flint have argued against transactional execution of system calls, reasoning that conflicts on kernel data structures will not only reduce concurrency in the transactional application, but will compromise performance isolation, resulting in overhead for all running applications [21].

Zilles and Baugh [20] and Moravan *et al.* [15], observing that it is untenable to execute syscalls transactionally, have proposed wrapping them in nontransactional regions. These regions, called *paused regions* in the former paper and *escape actions* in the latter, are blocks of code inside transactions that are not executed transactionally. Their memory footprints do not contribute to their enclosing transaction’s, and no automatic compensation is provided for them². We assume that all syscalls in transactions will be executed in such regions, and that consequently all syscalls will be side-effecting.

4.2 Experimental Method

We profiled our workloads using the Pin binary instrumentation tool [12] on an Intel Core 2 Duo dual-core processor. Because these programs are pthread-based, our Pin module tracked `pthread_mutex` acquires and releases and recorded I/O behavior, particularly syscalls, within these critical sections. For the purposes of our analysis, we assume that all of the critical sections we observed would be implemented as transactions, had the application been written for TM. We believe this is a fair assumption because, while locks can be used for mutual exclusion as well as isolated data access, our source inspection led us to believe that isolated access to data was a motivation for the subset of critical sections that we inspected in detail.

We measured durations using wall-clock time, using `gettimeofday()` with a granularity of one microsecond. When measuring time in Pin-instrumented code, it is necessary to consider the overhead incurred by the instrumentation itself. When tracking durations, we discounted time spent in instrumentation code by calculating the difference between the entrance to an instrumentation block and its exit, and subtracting that from the recorded duration of the instrumented critical section.

Critical sections, like transactions, may nest. For the purposes of our analysis, we only consider toplevel critical sections,

²However – as in *open nesting* [16] – it is possible to register blocks of abort (compensation) or commit code, for executing on the abort or on the commit of the parent transaction.

which we abbreviate to TCSs. Those TCSs which perform syscalls we abbreviate to syscalling-TCSs.

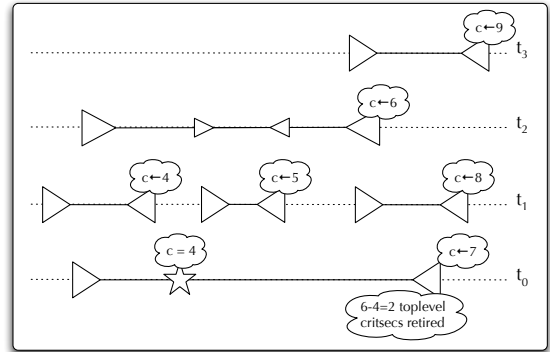


Figure 1: **A syscalling-TCS Overlapping with TCSs**
Right-pointing triangles are lock acquires; left-pointing triangles are lock releases. The star marks the first syscall in its syscalling-TCS. c is the global TCS retire counter.

In our analysis, we found it useful to determine the degree of concurrency, or *overlap*, experienced by syscalling-TCSs. This metric, which reflects the number of other TCSs that retired between a syscalling-TCS’s first syscall and its retirement, is gathered by a mechanism shown in Figure 1. We employ a global *toplevel critical section retire counter*, called c , which every TCS increments upon retiring. In the Figure, the syscalling-TCS on thread t_0 sees four TCSs retire in its lifetime. On its first syscall, it reads c , finding it to be 4. When the critical sections on threads t_1 and t_2 retire, they each increment c by one; the nested critical section on t_2 is not toplevel and so does not increment c . When the syscalling-TCS in t_0 closes, and before it increments c , it subtracts c ’s current value from the value read at its first syscall. Correspondingly, we say that the overlap of the syscalling critical section on thread t_0 is 2. Our dual-core processor may artificially limit the overlap we see, so to ensure that we were seeing as much overlap as possible, we inserted nanosleeps before every I/O operation (thus prompting the CPU to switch threads). When tracking TCS and syscalling-TCS durations, we discounted time spent in nanosleeps.

5 Results

We examine the syscalls made from within critical sections from two perspectives. First, we examine the syscalls themselves, to understand what is being called and with what frequency, and to discover what techniques can be employed by kernel or library developers to render these syscalls transaction-safe. Next, we examine the *context* of the syscalls, to discover the higher-level behavior of critical sections that contain syscalls.

Category of Syscall	Syscalls Seen in Critical Sections	Frequency in Critsecs	
		MySQL	Firefox
Time	gettimeofday, clock_gettime	3.91%	70.18%
Filesystem	read*, write*, open, close, lseek, access, dup, mkdir, ftruncate, fsync, writev, pread*, pwrite*, stat, fstat, fcntl, getdents, getcwd, fdatsync, mmap*, munmap*, mprotect*	53.79%	28.75%
Process Memory	brk, mmap*, munmap*, mprotect*	31.03%	0.32%
Process Maintenance	waitpid, clone, sched_setscheduler, sched_get_priority_max, sched_get_priority_min, rt_sigaction, rt_sigprocmask, tgkill	8.97%	0.32%
Communication	ioctl, socket, pipe, read*, write*, pread*, pwrite*	2.07%	0.40%
System Info	sysinfo, uname	0.23%	0.03%

Table 1: Syscalls Seen in Critical Sections *Six different categories of syscall were seen in critical sections in MySQL and Firefox; their dynamic distributions in the critical sections of each workload are shown at right. Communications syscalls, which are most difficult to make transaction-safe, are uncommon. Syscalls marked with asterisks belong to several categories.*

5.1 Syscall Sites

Previous work characterized syscalls inside critical sections for a selection of workloads, dividing them into `read` and `write` operations, and determining the frequency of critical sections performing them. [4] We delve deeper into the kinds of syscalls seen in our workloads, determining what calls are being made, what they are doing, and what minimum protection they require.

In Table 4.2, we list the syscalls dynamically detected in critical sections in our workloads, divided into six categories: filesystem, process memory, process maintenance, system info, time, and communication. Some of the syscalls listed belong to more than one category – for example, `read` may be applied to a file handle as well as to a socket. Such syscalls are marked with an asterisk. At the right side of the table, we show the relative dynamic frequency of each category of syscall in each workload. It is notable, in light of the observations we make in the next section, that very few communication syscalls are seen in either workload.

5.2 Syscall Protection – The Advantage of Compensation Code

As we have suggested, not all of the syscalls that we observed require the same treatment to become transaction-safe. We found four “protection classes” among the syscalls we observed:

- **Null compensation:** Some syscalls require no protection, as their speculative execution does not logically change system state. Time syscalls like `gettimeofday` fall in this class, as may syscalls like `pread`, which reads from a file without altering the file pointer. If side-effects are handled by forcing transactions to become non-speculative, this class of syscalls will not require that. If side-

effects are handled by compensation code, a null compensation block will be sufficient. Over 70% of the dynamic syscalls in Firefox critical sections fall into this category; under 10% of MySQL’s do.

- **Memory-fixup:** Many syscalls’ only side effect is a change of kernel state. For example, `lseek` does not affect a file directly, but instead adjusts a file offset pointer within a file handle data structure. Since kernel code is not executed transactionally, these actions must be considered as side-effects. If the TM system provides a mechanism for registering compensating code, then this work may be easily done speculatively; otherwise this side effect will necessitate either that the transaction go non-speculative, or that the call be deferred until commit.
- **Full compensation:** Many observed syscalls perform unprotected I/O actions, and will require “going non-speculative” or compensation code. For example, a transaction with an `open` call which creates a file will register a corresponding `unlink`, an `append` call might register a corresponding `truncate`³. If the filesystem has transactional support, then the compensation blocks might simply force a filesystem transaction abort.
- **Real actions:** A small minority of the syscalls we saw cannot be adequately compensated at the scope of the syscall. Some process maintenance syscalls like `tgkill`, and communications syscalls like `socket`, `pipe`, and `reads` or `writes` to sockets or pipes cannot be executed speculatively without knowledge of the *context* of the call. These syscalls are the most difficult

³While these compensations do not provide isolation from the rest of the system, it is not clear that this matters in most cases. For example, a critical section in MySQL responsible for database creation first created a directory for the database, then attempted to create an options file inside that directory. If the file creation failed, the directory was deleted again, violating system-wide isolation – but the programmers were willing to accept this result. A transactional filesystem can provide true isolation for filesystem side-effects, if needed.

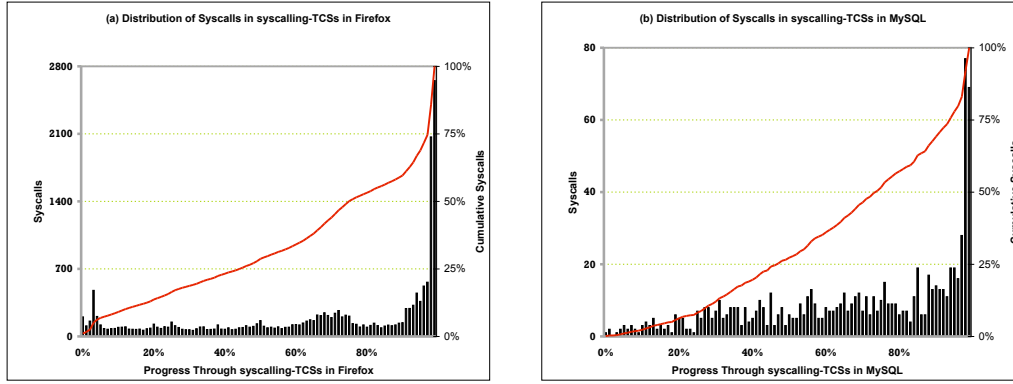


Figure 2: **Distribution of Syscalls across syscalling-TCSs**

Syscalls are distributed throughout syscalling-TCSs, and are more frequent towards the end of the critical sections.

Aggregated results for all syscalling-TCSs in (a) Firefox, (b) MySQL. Bars are per-bin with axis on left; line is cumulative with axis on right.

to encapsulate so that programmers at higher levels of abstraction may use them speculatively, but some possibilities exist. Buffering support in the manner of Re-ViveI/O may be employed [17], but unlike that work, the amount that may need buffering is unbounded. Alternatively, if it is possible, compensation may be provided at a higher level of abstraction by the application or library developer. For example, a network program may use a transaction-safe protocol, in which messages may be tentatively issued and revoked later – or it might not, and this information would not be available at the level of the system call. In these cases, the only choice is to wait until the transaction is nonspeculative before executing the syscalls – or permitting compensation code to be registered at a higher scope. This class corresponds to the Communication category described in Table 4.2, as well as some process management syscalls like `clone` and `tgkill`. It comprises about 7% of the syscalls in MySQL’s critical sections, but a minuscule component of Firefox’s.

Importantly, all these categories except for the last may be compensated for at the level of the syscall – and the last category represents very few of the dynamic syscalls we encountered. It is clear that with adequate compensation the bulk of dynamic syscalls can be rendered transaction-safe, and thus speculatively executable. For the workloads we examined, written in C/C++, this could be accomplished simply by providing compensation code in the system library (e.g. `libc`); if this were done, our workloads could be transactified with nearly all syscalls in transactions handled transparently to the application developers.

5.3 Syscall Context

We have examined the types of syscalls executed from critical sections in our workloads; now we examine their context, exploring the structure of critical sections which perform syscalls. We will attempt to characterize side-effecting transactions by their frequency, by the distribution of syscalls within them, by their (temporal) length, and by the degree of concurrency they expose.

Previous research on other applications has shown that syscalling critical sections are dynamically very rare in multithreaded workloads [4]. We find the same: in Firefox, only 0.71% of dynamic critical sections issue syscalls; in MySQL, the proportion is even smaller: only 0.02%. However, further examination of these critical sections shows that it may not be wise to dismiss them as too rare to matter.

Figure 3 shows the approximate durations, in μ seconds, of toplevel critical sections – both those which execute syscalls (the syscalling-TCSs) and those which do not – in our workloads. In both workloads, we find that critical sections performing syscalls tend to be much longer than those which do not. To what degree this reflects an intrinsic quality of atomic regions which perform syscall, and to what degree it reflects the cost of switching into kernel mode, matters less to us than the fact that critical sections performing syscalls tend to be quite long. The longer transactions last, the more chance they have to affect the performance of other transactions in the same application

5.4 The Applicability of Deferral

In Figure 2, we show where, in the progress of syscalling-TCSs in our workloads, syscalls are executed. In these graphs, every syscall executed within a syscalling-TCS (including

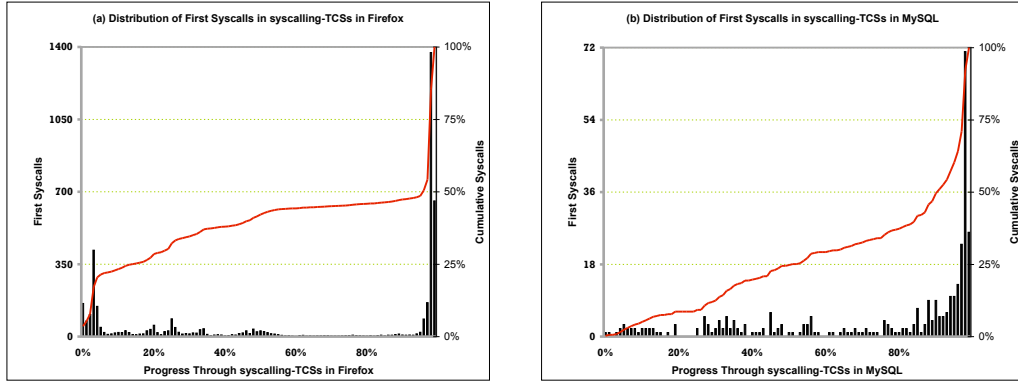


Figure 4: **Distribution of First Syscalls across Critical Sections**

A significant fraction of first syscalls occur well before the end of the critical section. Aggregated results for all critical sections in (a) Firefox, (b) MySQL. Bars are per-bin with axis on left; line is cumulative with axis on right.

syscalls indirectly called by nested children of the syscalling-TCS) increments the bar corresponding to that syscall’s position in the lifetime of its syscalling-TCS. These graphs show that while syscalls do tend frequently to be positioned near the end of their syscalling-TCSs, they exist in significant numbers throughout the lives of syscalling-TCSs. This has several ramifications for the TM proposals reviewed in Section 3. For techniques which defer syscalls until their transaction is validated⁴, the earlier a syscall happens, the longer it must be deferred, and the more significant its implicit reordering becomes – a problem both for the programmer and for the static analysis tools the compiler might use to choose critical sections to defer. Furthermore, as an operation may not be moved after any of its consumers, any syscall in a TCS which produces values that are consumed later in the TCS cannot be deferred. We analyzed syscalling-TCSs responsible for 90% of the dynamic syscalling-TCS instances in our benchmarks, and found that over 96% of those in MySQL, and 100% of those in Firefox, consumed the result of the first syscall in the TCS – suggesting that deferral will not apply in the preponderance of cases. For the technique of forcing transactions to “go non-speculative” prior to executing any syscall, a more serious problem awaits.

5.5 The Overhead of “Going Non-speculative”

In Figure 4, we show the distribution of the positions of *first* syscalls within the durations of their syscalling-TCSs. In these graphs, the first syscall executed within a syscalling-TCS (including syscalls indirectly called by nested children of the syscalling-TCS) increments the bar corresponding to that syscall’s position in the lifetime of its syscalling-TCS. The figure shows that while a significant number of syscalling-TCSs have their first syscall near the end of their lives – over 50% in the last 10% of Firefox syscalling-TCSs – many execute their first syscall relatively early. If transactions attempting

syscalls must first “go non-speculative”, then from that point – the point of the first syscall – until their commit, no other transaction may become non-speculative. This Figure shows that at least one third of syscalling-TCSs in both workloads see their first syscall before they are halfway finished. As the barrier to concurrency is a problem proportional to the length of syscalling-TCSs, which, as shown in Figure 3, tend large, this represents considerable potential for even a relatively small number of toplevel side-effecting transactions to impact concurrency throughout applications.

These large transactions translate to a significant degree of overlap (described in Section 4.2.) The overlaps of syscalling-TCSs in our workloads are shown in the black plots in Figure 5. While a majority of syscalling-TCSs overlap with no other TCS – two-thirds in Firefox and half in MySQL – a significant minority have substantial overlap. 15% of the syscalling-TCSs in Firefox overlap with 360 or more TCSs; if the presence of a non-speculative transaction precludes any other transactions from committing (*e.g.* by holding a “commit” token), this betokens a significant loss of concurrency throughout both our applications.

Blundell *et al.* suggest that the non-speculative transaction (the *unrestricted* transaction in their parlance) need not block speculative (*restricted*) transactions from retiring – as long as there is only one non-speculative transaction active at a time, and it may never be aborted. [3] Their *restricted* transactions are not only constrained from “going non-speculative”, but are also bounded in time and memory footprint. However, the memory and time bounds are not strictly required; there is no reason that speculative transactions of any size or duration might not retire even while another transaction is non-speculative, so long as the speculative transactions do not conflict with the non-speculative. In this case, it is instructive to consider, as a lower bound to syscalling-TCS overlap, the overlap that syscalling-TCSs have with other syscalling-TCSs. This metric, measured in the grey plots in Figure 5, is similar to that shown in Figure 1

⁴That is, known to be safe to commit, and correspondingly non-speculative

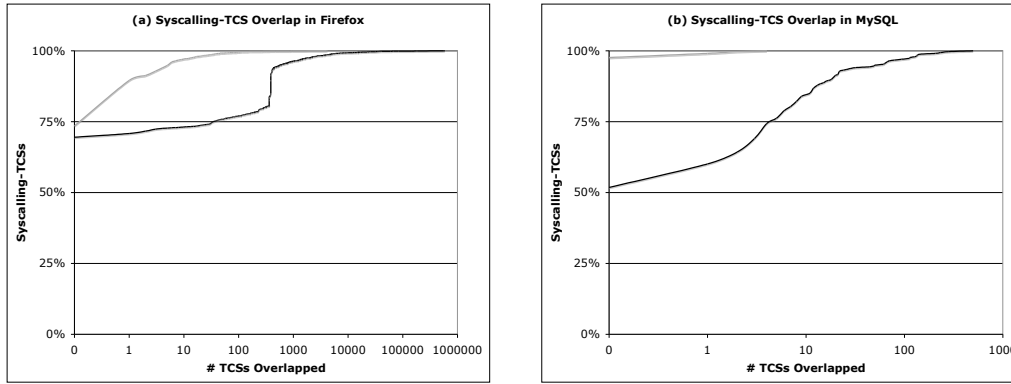


Figure 5: **Degree of Overlap of Toplevel Syscalling Critical Sections (syscalling-TCSs) and Toplevel Critical Sections (TCSs)**

All syscalling-TCSs in (a) Firefox, (b) MySQL. This cumulative plot shows (black line) how many syscalling-TCSs (y-axis) overlap (x-axis) or fewer TCS; (grey line) how many syscalling-TCSs (y-axis) overlap (x-axis) or fewer other syscalling-TCSs

(Section 4.2), except that not every retiring TCS increments the global TCS retire counter – only retiring syscalling-TCSs may. As only one syscalling-TCS is allowed to be nonspeculative at a time, the overlapped critical sections in the grey plots in Figure 5 cannot be executed concurrently under a regime in which syscalling-TCSs “go nonspeculative” – but the loss of concurrency is much less than that seen in the black plots in Figure 5: over 93% of Firefox syscalling-TCSs overlap with 10 or fewer other syscalling-TCSs, and only 2% of MySQL syscalling-TCSs overlap with any other syscalling-TCSs at all, none with more than 4 others.

We expect these results to be a lower bound of the actual concurrency available in transactional versions of these workloads. Atomic regions which, in lock-based code, would be guarded by different locks, will, of course, when transactional, not conflict (for if they did, the lock-based code would have race conditions.) However, some atomic regions which in lock-based code would be guarded by the same lock might as transactions not conflict.

6 Conclusion

While transactional memory is a promising technique for achieving concurrency in synchronized code, two of its oft-cited advantages – composability and programmability – are compromised by the difficulty of speculatively executing side-effecting code. In this paper, we have examined the side-effecting operations, particularly the syscalls, performed in critical sections in two large, multithreaded workloads. We have classified the kinds of syscalls thus performed, noting that the presence of correct compensation code and a transactional filesystem permit nearly all syscalls to be executed speculatively. Transaction-safe system libraries, linked to transactional filesystems, could enable application programmers to

speculatively invoke, directly or through composition, nearly every syscall.

We have also examined the contexts of syscalls in critical sections in our workloads, observing that those critical sections which do perform syscalls do so throughout their lifetimes, and that there is a strong tendency for results of syscalls in critical sections to be used within those critical sections – which may limit the usefulness of deferring I/O until the end of its enclosing transaction – and that those lifetimes tend to be quite long compared to other critical sections. Considering the effect that long syscalling transactions might have on overall concurrency available in applications, we examined the amount of overlap syscalling critical sections have with other critical sections. We observe that the technique of “going nonspeculative” results in substantial loss of concurrency when not applied carefully, but that this loss can be dramatically reduced if nontransactional transactions only preclude the retirement of other conflicting transactions, or other transactions which perform syscalls.

References

- [1] Sysbench: a system performance benchmark.
- [2] C. S. Ananian, K. Asanović, B. C. Kuzmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of HPCA-XI*, Feb. 2005.
- [3] C. Blundell, E. C. Lewis, , and M. M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, July 2006.
- [4] J. Chung, H. Chafi, A. McDonald, C. C. Minh, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common

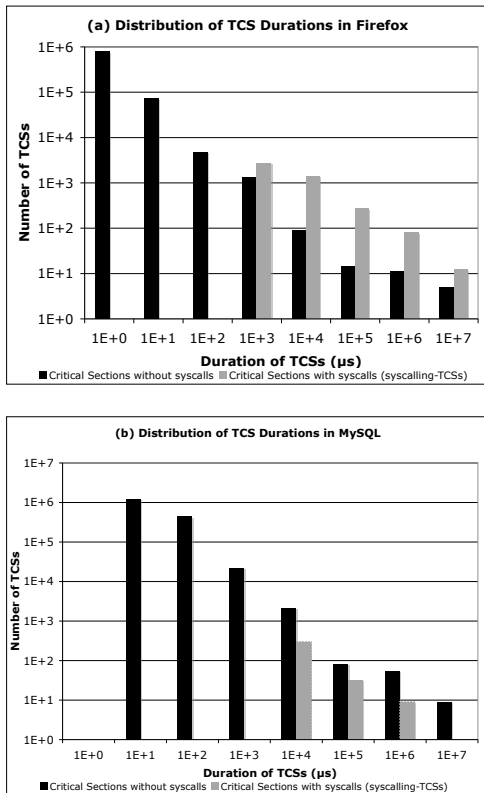


Figure 3: **Distribution of syscalling-TCS Durations (in μsec)**

TCSs with syscalls trend much larger than those without. Black bars are TCSs without syscalls; grey bars are syscalling-TCSs. Results for (a) Firefox, (b) MySQL

case transactional behavior of multithreaded programs. In *Proceedings of HPCA-XII*, Feb. 2006.

- [5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA-31*, pages 102–113, June 2004.
- [7] T. Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.
- [8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Principles and Practice of Parallel Programming (PPOPP)*, 2005.
- [9] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Symposium on Principles of Distributed Computing (PODC)*, 2003.
- [10] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA-20*, pages 289–300, May 1993.
- [11] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, Dec. 2006.
- [12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.
- [13] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA-33*, June 2006.
- [14] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of HPCA-XII*, Feb. 2006.
- [15] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. *SIGOPS Oper. Syst. Rev.*, 40(5):359–370, 2006.
- [16] J. E. B. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings of the workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.
- [17] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. Revivei/o: Efficient handling of i/o in highly-available rollback-recovery servers. In *Proceedings of HPCA-XII*, Feb. 2006.
- [18] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA-32*, June 2005.
- [19] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
- [20] C. Zilles and L. Baugh. Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions. In *First ACM SIGPLAN Workshop on Transactional Computing*, June 2006.
- [21] C. Zilles and D. Flint. Challenges to Providing Performance Isolation in Transactional Memories. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, pages 48–55, June 2005.