Architecture Support for Parallelization using Thread Level Speculation (TLS) with an Implementation in GEM5

A Project Report

submitted by

SRISESHAN SRIKANTH

in partial fulfilment of the requirements
for the award of the degrees of

MASTER OF TECHNOLOGY
and
BACHELOR OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2014

THESIS CERTIFICATE

This is to certify that the thesis titled Architecture Support for Parallelization using Thread Level Speculation (TLS) with an Implementation in GEM5, submitted by **Sriseshan Srikanth**, to the Indian Institute of Technology, Madras, for the award of the degrees of **Master of Technology and Bachelor of Technology**, is a bona fide record of the project work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Shankar Balachandran
Dual Degree Project Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: May 4, 2014

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my project guide Prof. Shankar Balachandran, Dept. of Computer Science and Engineering for his wholesome support, motivation and guidance throughout the course of this work. I would like to thank Prof. José Nelson Amaral, Dept. of Computing Science, University of Alberta for his mentorship as well as his insight on thread level speculation. I am ever grateful for the time they took in guiding and helping me towards a career in research.

I would like to thank my co-guide, Prof. Nitin Chandrachoodan, and Prof. K Sridharan, Dept. of Electrical Engineering for our informal discussions on architecture. Also, I would like to thank the Dept. of Electrical Engineering and the P.G. Senapathy Centre for Computing Resources at IIT Madras, University of Alberta and the MITACS Globalink fellowship program for facilitating my work.

An extremely special thanks to my dear family and friends for understanding and supporting me. I really appreciate everything they have done for me, without me having to ask for it even.

ABSTRACT

Out of order RISC processors employ dynamic sheduling and register renaming mechanisms to reorder ALU type instructions, thereby extracting more Instruction Level Parallelism (ILP) from a thread. ILP is further boosted by using data speculation mechanisms, wherein instructions from a sequential instruction stream are allowed to be reordered irrespective of interspersed loads and stores. However, on a multi-core system, memory accesses are not exclusive to a given core; they are often shared by multiple threads running on different cores. Therefore, it becomes essential to exploit Thread Level Parallelism (TLP) to improve performance.

Thread Level Speculation (TLS), as the name suggests, provides a mechanism by which data speculation can be done across threads, without regard to any dependencies that may exist among them, and at the same time, ensuring correct sequential semantics as assumed by the program. The prime utility of such a framework is to help extract parallelism from applications where compilers conservatively render sequential code when they find it difficult, if not impossible, to statically disambiguate arbitrary memory references. This work presents a detailed architectural design for TLS, an implementation on a widely used open-source cycle-accurate simulator known as *gem5*, and a demonstration showing the utility of using TLS where conventional auto-parallelizers fail. For explicitly parallelized code, this implementation reports a near identical performance profile whether or not the TLS feature is used, thus indicating negligible overhead. Based on insights gained throughout this work and through the analysis of this implementation itself, several inputs regarding development of compiler support to best leverage TLS are given, in addition to possible architectural extensions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multicore systems are ubiquitous in today's world and to fully utilize them, it is important to extract as much parallelism as possible from applications, the onus of which lies largely with the programmer. When coupled with the fact that legacy code cannot be dispensed with, this has resulted in an under-utilization of parallel hardware that is available, even in the presence of auto-parallelizing compilers. Significant progress has been made in auto-parallelizing regular numeric applications; however, the impact of auto-parallelizers has been limited mainly because, to schedule a given loop for parallelization, they require a guarantee that the said loop is actually independent.

Thread Level Speculation (TLS), also known as Speculative Multi-Threading (SpMT), is a hardware and/or software approach aimed at improving this scenario with little or no burden on the programmer. By means of careful book-keeping, TLS is designed to ensure that correct execution of code is guaranteed, and this enables compilers to optimistically render parallel code even in presence of inter-iteration *may* dependencies. In a broader sense, TLS and Transactional Memory (TM) share common design ideologies.

The apparent advantage of such a framework is better appreciated by considering the case of *pseudo* parallel loops. For example, when all but a few iterations of a loop are devoid of data dependencies, or, when only strided dependencies (with a stride of atleast 2) are present, it is clear that parallelism exists for such a loop at a fine-grained level. However, a traditional compiler would conservatively analyze this loop to consist of dependencies, and hence this loop ends up being run sequentially. With TLS, however, various iterations of such a loop can indeed be speculatively executed in parallel. In case a data dependence (*conflict*) is detected at runtime, TLS employs a fallback, retry or synchronization mechanism to ensure correctness of data.

The deficiency of non-TLS auto-parallelizers is seen especially in the presence of pointers and input-dependent parameters. In such cases, even embarrassingly parallel loops are classified as having *may* dependencies, thereby rendering them unfit for parallelization. Using TLS in such scenarios helps unlock potential parallelism.

However, it must be noted that though TLS provides a guarantee on correctness, it comes at the cost of *misspeculation*. *I.e.,* in the event where a conflict is actually detected at runtime, any mechanism that TLS uses to ensure correctness of data is bound to lower performance because of redundancy in execution. In other words, it is imprudent to use TLS in cases where a loop is inherently sequential. Therefore, it can be seen that TLS provides a guarantee only on correctness of data - not on performance. It simply provides a framework by which it is relatively easier for the compiler/end-user to extract parallelism from an application as opposed to using a traditional setup.

Chapter [2] presents more details about TLS in general and why contributions of this work are important. In this work, the design of the TLS framework is loosely based on an idea that was first presented in [1]. Implementation and analysis of this design is done on *gem5* [2], an open-source and modular simulator platform that is widely used in computer systems architecture research today.

The main contributions of this work are as follows:

- A design that enables speculative buffering of data and supports runtime conflict detection.
- An X86 based *gem5* implementation that can easily be extended to other architectures, as well as to support Transactional Memory (TM).
- A preliminary analysis that demonstrates the utility of the implementation, and provides insight towards development of compiler support to fully leverage TLS. Apart from providing speedup in cases where auto-parallelizers fail, this implementation shows a near identical performance profile to that of a conventional non-TLS architecture while running explicitly parallelized code.
- An in-depth description of the design and implementation, to aid further development of this framework.

Throughout this report, the terms *core* and *processor* are used interchangeably. The term *thread* may refer to either a hardware or a software thread as a one-one mapping is assumed.

The rest of this report is organized as follows: Chapter [2] briefly explains the idea of TLS, its history and the motivation behind this work. Chapters [3] and [4] provide, respectively, details of the designed architecture and the *gem5* implementation. Two chapters of Appendix are written to further elaborate on these two chapters. Chapter [5] demonstrates the utility of this implementation and Chapter [6] summarizes the goals achieved and the insights obtained during the course of this work, in addition to providing directions for future research.

Chapter 2

Background

2.1 Thread Level Speculation

As mentioned earlier, Thread Level Speculation enables parallelization of code that contains *may* dependence. With the hardware and/or software guaranteeing correctness of data, the compiler can optimistically parallelize code without having to conservatively confirm the absence of data dependencies.

The concepts of *commit* and *squash* are extended from the context of speculative execution that occurs at the instruction level. When a thread completes its speculative execution without running into any data dependence violations or any irrevocable instructions (such as I/O), it can be *committed* as this speculative execution is deemed correct. On the other hand, upon detection of a conflict, such a thread - along with all other threads that may be affected by it - must be *squashed* as this speculative execution is deemed incorrect, from the standpoint of sequential semantics assumed by the program.

This work adopts a hardware-software approach to realize TLS; a high level description of which is as follows: The hardware maintains a global time ordering in a first come first serve manner of threads, to match the *spawn* order as determined by the software. The cache system is modified to enable speculative buffering of data and to detect conflicts based on this order, by extending a directory based MESI cache coherence protocol. Any given speculative thread can *commit* results from its speculative execution only after all its earlier threads have committed successfully. This way, if there is a conflict, all speculative threads newer (younger) than the thread that detected this conflict are *squashed*. Such *squash*es - necessary to protect data from corruption - are communicated back to the runtime, which ensures forward progress by re-executing such threads.

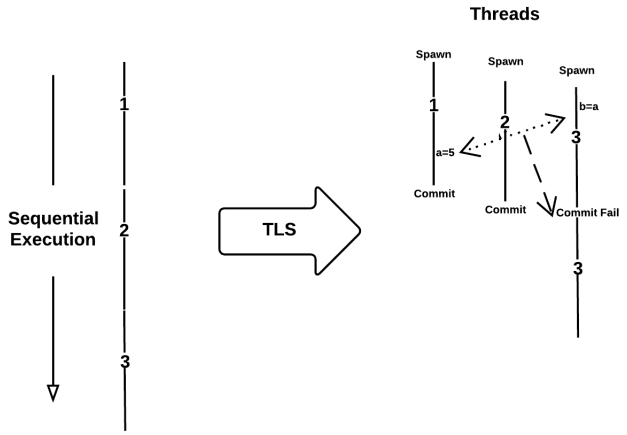


Figure 2.1: Illustration of TLS in the case of a RAW dependency

To illustrate, consider the example shown in Figure 2.1. There is a Read-After-Write (RAW) dependency between threads 1 and 3 through variable a. In the event where a is pointed to by a pointer p in thread 1 and by a pointer q in thread 3, where p and q may point to the same memory location, traditional auto-parallelizing compilers classify this as a *may* dependence, making parallelization impossible for the entire loop. But TLS support makes exploitation of parallelization within a part of the loop a possibility.

2.2 Evolution

The idea of performing memory loads and stores speculatively with general purpose programs in mind has been around for roughly two decades. It was first conceived as a means to further exploit Instruction Level Parallelism (ILP) via the use of Address Resolution Buffers (ARB) [4]. ARB achieved speculative versioning by buffering all versions of a given memory location as a separate entry [5]. The biggest drawback of this approach was that the ARB was a centralized (shared among processors) buffer, which meant that it ran into latency and bandwidth bottlenecks. Naturally, later approaches made the speculative versioning more distributed and sought to integrate it with the cache system.

The concept of Speculative Versioning Cache (SVC) [6] was built upon tradi-

tional bus based snooping cache coherence. In this approach, each line of a private cache was extended to indicate if it had been speculatively loaded, and to include a pointer that identified the processor containing the next version of that line. A separate, dedicated Version Control Logic (VCL) was used to provide responses to the private caches. These responses were given in a manner similar to that of the disambiguation logic that ARB employed, *i.e.,* by searching previous versions in the case of a load, and later versions in the case of a store. A similar scheme [7] was concurrently proposed wherein a simpler coherence protocol was adopted, however, at the expense of bursty traffic upon completion of each speculative context - as was the case with ARB. Unlike SVC, this scheme relied completely on software for task assignment to a processor.

Another approach [8] to TLS made use of a dedicated speculative co-processor that helped execute a number of software speculation control handlers, thereby lying somewhat in between the previous two schemes. However, to simplify cache coherence, write through caches were employed in this scheme.

An orthogonal approach [9] was developed with an aim of simplifying the conceptual and implementation complexity of such TLS (and TM) mechanisms. The addresses accessed by a thread were compactly encoded using a Bloom filter based hash, resulting in a superset representation. This enabled easy implementation of operations on groups of addresses.

Over the years, these groups proposed improved versions of their schemes, but the basic ideas have remained unchanged. Also, several software-only implementations like [10, 11, 12] have been proposed, but as can be expected, these are very slow when compared to approaches that used hardware support.

Recently, IBM released a system - BlueGene Q [13, 14] - that has inbuilt support for TLS [15]. In this architecture, speculative versions of a line are placed in separate ways of the directory set that stores the non-speculative version. To distinguish versions, additional tags are used in the directory.

As mentioned earlier, this work adopts a hardware-software approach to TLS. It may be observed that most of the book-keeping techniques that have been pro-

posed for detection of conflicts, like the VCL of the SVC scheme, give a directory-esque structure to the design. A directory based coherence protocol lends itself naturally for such functionality, and is hence adopted in this proposed design. Moreover, similar techniques that are used to make a directory more distributed and scalable, can be applied here as well.

2.3 Motivation

Silicon technology has advanced to a stage where a hardware implementation of TLS is now commercially viable - as evidenced by BlueGene Q, which however, has a rather under-utilized TLS framework. Also, there is a lack of a well-supported platform for the community to collaboratively research TLS. These points are explained in this Section, serving as motivation for this work.

As part of a related work [16] that involved writing a software framework for auto-parallelization using TLS on BlueGene Q, it was necessary to gauge the overhead caused due to TLS. For this purpose, the following loop was run after compiling for TLS using a BG/Q specific C compiler - *bgxlc_r*:

```
for (i = WINDOW; i < SIZE; i++)
    a[i] = a[i-WINDOW] + 1;
```

The intention behind this loop is that the degree of dependence changes with the value of *WINDOW*, thus causing different conflict probabilities. For example, if *WINDOW* is 0, then the loop is perfectly independent and for several non-zero values, conflicts are expected. Given that BG/Q maintains speculative states in the L2 cache and that the spawn policy used is that of *OpenMP* (*bgxlc_r* uses the *OpenMP* runtime for parallelism), a sweep from 0 to *SIZE*/10 was done for *WINDOW*, with steps of varied granularity. A *SIZE* of 10000000 was chosen so that when TLS runs long enough, the conflict overhead is not masked. Upon collecting statistics, as given in Table [2.1], the only monotonic relationship observed was that between L2 hits and execution time (cycles). The most puzzling aspect was

WINDOW	Rollbacks	L1p Misses	L2 Hits	L2 Misses	Cycles
0	585	81192	22958266	109512803	638504716
1	585	82795	25878786	109268345	920234122
2	588	90346	23288451	96260552	849992884
3	586	4050901	22460165	111945889	636492262
4	585	4059102	22632073	112311636	653979778
5	586	4079870	26683489	112933698	977863450
6	586	4070801	27767357	113041372	1075979998
7	586	4088353	27578418	113515836	1049858014
10	586	4081335	26828284	112952959	990203554
100	585	90071	25947190	108819788	938475532
1000	586	93373	21983026	110167998	624440650
10000	588	90806	26079182	104113107	948449626
100000	587	91694	21691725	78139164	602617072
1000000	585	90716	23893375	61895535	860489428

Table 2.1: Counter values measured upon TLS execution of a simple loop with varying degrees of dependence. A rollback occurs when speculative execution encounters a conflict.

that the measured number of rollbacks was nearly constant over all the values for *WINDOW*.

Attempts to ensure that the compiler doesn't optimize away the dependencies were made by declaring a as a pointer instead of an array and by wrapping the above loads and stores into separate functions. Also, the initial values of a and the value of *WINDOW* were randomized so that information about these was available only at runtime. Yet, the measured number of rollbacks remained nearly constant, at ≈586, which is less than 0.01% of *SIZE*, where *SIZE* is roughly the total number of iterations. If this was a large value, then it could have been be argued that the granularity of the underlying conflict detection in the L2 cache line was too coarse, thus making many false positives. However, as relatively small number of rollbacks were observed, two possible explanations exist: The compiler serialized the loop or unrolled it effectively to hide the dependency, or, the L2 cache line was evicted prematurely, *i.e.*, before a conflict could be detected. The *qnounroll* compiler option was tried, but no change in behaviour was observed. Thus, either the compiler chose to spawn very few iterations as parallel TLS threads; thereby serializing the loop to execute in large chunks, or, the spawn policy of *OpenMP* was suboptimal; causing frequent cache eviction.

Clearly, a deeper understanding of the underlying system was necessary to develop an auto-parallelizing framework that leverages the utility of TLS. This indicated a necessity to avail of a platform where architectures and various compiler designs for TLS was supported. No other publicly available cycle accurate simulator apart from *SESC* [18] has incorporated support for TLS. Given that the *gem5* simulator is very modular, flexible and open-source, is widely used and supported actively by the architectural research community [19], and that *SESC* supports the *MIPS* architecture alone and does not provide a means of booting a full fledged system, this work uses *gem5* as its underlying implementation base.

Chapter 3

Design

It is assumed that there are N cores with a private L1 cache each and a shared L2 cache, which is the LLC (Last Level Cache) and is associated with the directory. Of course, this can be extended to more levels of cache, but this implementation uses a two level directory based cache hierarchy.

Sections [3.1], [3.2] and [3.3] elucidate necessary changes that have to be made at the Core, Private Cache and Directory levels respectively. The approach followed is that of strictly inclusive caching and of allowing only single speculative writer. Reasons for choosing these, and strategies for eviction and commit are discussed in Section [3.4].

Two important assumptions are made. One is that instruction blocks cannot be speculatively modified incorrectly, as, invalidating such lines as a result of squash would render faulty program flow. This will be clearer upon considering an end-to-end implementation of this design, *i.e.,* by Section [4.4]. The second assumption is that the code has neither non-speculative nor irrevocable (such as I/O) instructions within the limits of a TLS section.

3.1 Core

Unlike conventional speculative execution (instruction-level), thread level speculation, as indicated by the name, requires knowledge of the execution status of instructions - loads and stores, in particular - from other threads. Moreover, TLS concerns itself only with memory accesses, and, the cache hierarchy has more control over and access to the higher level notion of a thread. Therefore, introducing buffers for speculative results within the processor is meaningless. In other words, if one were to use such buffers, inter-core query messages would end up eating away the entire network bandwidth.

TLS	SQ	Processor State
0	0	Processor is executing in Normal mode
1	0	Processor is executing in TLS mode. No squash detected so far
1	1	Processor is executing in TLS mode. Atleast 1 squash has been detected
0	1	Processor is in a transient state between TLS mode and Normal mode. Goes to Normal mode once invalidation of (mis)speculative addresses is done

Table 3.1: TLS Bits per Core

With bookkeeping for TLS being handled by the cache subsystem, all that is left to be desired of a core is a means to interact with the runtime. This interaction is for requests from the runtime to the core to start or stop TLS, and, responses back to the runtime indicating their success or failure. This can be succinctly achieved by means of two bits: *TLS* and *SQ*. The *TLS* bit indicates if the core is running in TLS mode, *i.e.,* if it is issuing speculative loads and stores. The *SQ* bit indicates if atleast one squash has been communicated by the cache subsystem to the core in the current (or immediately previous) TLS execution. Their meaning when used together, is explained in Table [3.1].

These bits could be part of a non general purpose register like a control register. Methods to read and manipulate these bits in the context of x86_64 are explained in Sections [4.1] and [4.4].

These bits are propagated to the sequencer at the L1 cache controller along with the other lines that indicate whether the request is a load or a store or an instruction fetch etc.. Subsequent interpretation of these bits by the cache subsystem is discussed in Section [4.3].

3.1.1 Design Complexity

In terms of storage, an overhead of two bits per core is incurred for the *TLS* and *SQ* bits. In terms of routing, two extra lines are required between the core and the cache subsystem. The timing overhead caused by this scheme on a memory access path is limited to a few multiplexers (and demultiplexers) that would be necessary for reading and/or writing the two bits, and in asserting the two lines

as part of the memory request. Though the exact number of these muxes (and demuxes) may vary depending upon the implementation, it is obvious that the clock frequency of the core would remain unaffected by these.

3.2 L1 Cache

Upon receipt from the core, complete with the *TLS* and *SQ* lines, the request gets decoded and the cache coherence protocol state machine triggers a transition accordingly. The coherence protocol, itself, is an extension of the MESI directory protocol. This extension serves two primary purposes:

- To enable bookkeeping for TLS, *i.e.,* to enable speculative buffering of data, and,
- To enable detection of conflicts such as *RAW* (Read-After-Write or Flow) and *WAW* (Write-After-Write or Output), so that a *squash* can be issued.

Each thread (or equivalently each core and its private L1 cache) is assigned a particular *specID*. The notions of earlier/older than (<) and later/younger than (>) are based on the relative values of *specID*s. This way, a conflict can be detected when a thread with a higher *specID* tries to (speculatively) read or write into a location that is being written into by a thread with a lower *specID.* The exact mechanism by which assignment and conflict detection are done is explained in Section [3.3]; the directory maintains all these specIDs.

The resulting protocol, thus, adds 3 new states to the design to indicate TLS: *SpM, SpE* and *SpS*. The meaning of each of these 7 stable states is explained in Table [3.2].

A simplified state diagram with just the stable states and their transitions to events triggered by the processor as well as other coherence messages is depicted by Figure [3.1]. A detailed state transition table complete with all transient states and actions is available in Appendix [A.1]. This detailed table is designed (especially *w.r.t.* coherence messages) in a manner that makes it conducive to debugging.

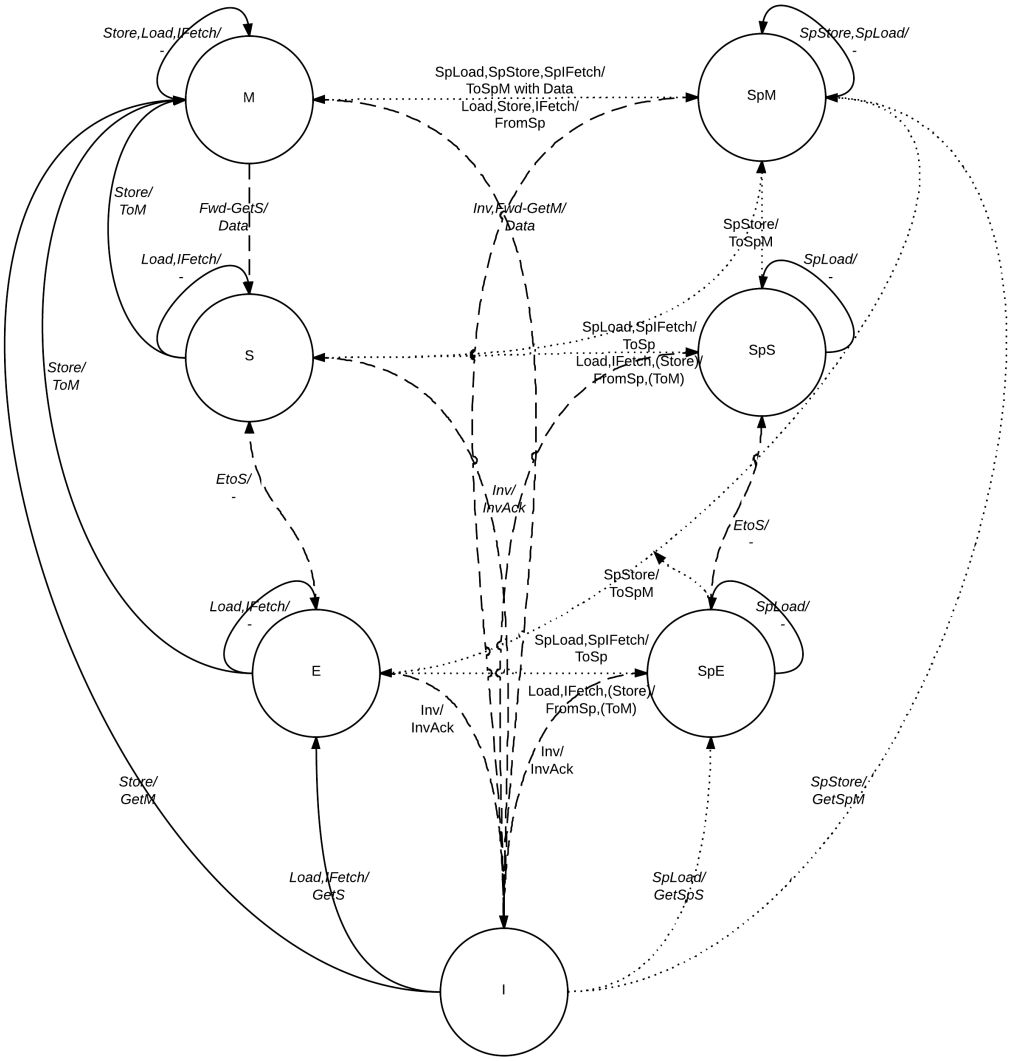


Figure 3.1: Simplified TLS Cache Coherence State Machine - L1 Cache
The solid lines represent processor generated requests, the dashed lines represent directory or inter-cache messages, the dotted lines indicate messages generated due to TLS alone.

State	Meaning
I	Invalid block
E	Block contains clean data. This cache is the owner, read access only
M	Block contains dirty data. This cache is the owner, read-write access
S	Block contains clean data. This cache is not the owner, read access only
SpE	Block contains clean data. This cache is the owner, read access only. This core may be running in TLS mode*
SpM	Block contains dirty data. This cache is the owner, read-write access. This core may be running in TLS mode*
SpS	Block contains clean data. This cache is not the owner, read access only. This core may be running in TLS mode*

Table 3.2: L1 Stable States. $*$: The clause *may be* is better appreciated in Subsection [3.4.4].

Also, a single bit *isSp* is used to indicate if the core is running in TLS mode or not. This bit is set as soon as a TLS request is decoded, which indicates that the core has entered into TLS mode, and is reset only upon receipt of a non-TLS request, or in the case of a *squash*.

3.2.1 Design Complexity

In terms of storage, apart from the single *isSp* bit per L1 cache, there is one extra bit per cache block to accommodate the extra base states. The bit encoding of these states is shown in Table [3.3].

Significant complexity arises from the *MSHR* (Miss Status Handling Registers that handle transient states for the cache) and cache controller logic to handle new messages. As this design is currently intended for a simulator implementation (as opposed to synthesis of real hardware), redundant messages exist to make debugging easier.

States	Bits			
	Valid	Exclusive	Dirty	TLS
I	0	×	×	×
E	1	1	0	0
M	1	1	1	0
S	1	0	0	0
SpE	1	1	0	1
SpM	1	1	1	1
SpS	1	0	0	1

Table 3.3: Bit encoding of stable states at each L1 cache block

3.3 Directory

Under a basic and non-TLS scheme, the directory contains - for each valid block - the directory state, the owner and an *isShared* bit vector that indicates which of the L1 caches share the block. For TLS, in addition to the above, a single *isSp* vector is needed. This vector stores 0 for non-TLS cores and a *specID* ($>= 1$) for TLS cores. Reasons for storing this *isSp* vector in the directory instead of the L1 caches are as follows:

- Every message need not have the *specID* piggy backed onto it.
- The directory can make decisions (like detect conflicts) and take actions (like issue *squash*) without redundant messages involving L1 caches, which would thus unnecessarily have duplicate logic.
- It is eventually possible to allow replacement of speculative blocks. This would allow threads with speculative contexts larger than the L1 cache capacity to speculatively execute, thereby allowing context switches without having to *squash* them. This is explained in Subsection [3.4.2].

As L1 caches have TLS data encoded within their states, the states in the directory state machine remain unchanged from the MESI protocol. However, the meaning of each state is slightly modified to account for the additional L1 states. This is shown in Table [3.4]. A detailed state transition table complete with all transient states and actions is available in Appendix [A.1]. As mentioned earlier, this detailed table is designed (especially *w.r.t.* coherence messages) in a manner that makes it conducive to debugging.

State	Meaning
I	Invalid Block. Not present in any L1 cache.
E	Block contains clean data. Owned by an L1 cache.
M	Block contains dirty data. Owned by directory if not present in any L1 cache. Owned by an L1 cache if present in atleast 1 L1 cache. If present in >1 L1 caches, owner is in *SpM* and other L1 caches that contain it share it, possibly speculatively, with a *specID* less than that of the owner.
S	Possibly dirty if transitioned from *M*. Owned by directory. Present in zero or more L1 caches.

Table 3.4: Directory Stable States

3.3.1 Design Complexity

The total number of *specID*s in any scenario can be limited to $2N$-1. This is because *commit* is done in-order and a round-robin *specID* assignment can be done to the cores. To elaborate, upon receipt of a TLS request from an L1 cache, if that core is not yet marked with a *specID* (or, equivalently, if that core has a *specID* of zero), it is assigned a *specID* with a value equal to {max(*specID*s allocated so far)+1}. As *commit* is done in increasing order of *specID*s, the assigned *specID*s go from 0 to the current maximum *specID*.

Without loss of generality, consider the case where core i has been assigned with *specID* i, where i goes from 1 to N. At this point, each of the N cores is executing in TLS mode, *i.e.,* if any more threads are to execute speculatively, one of these cores must *commit*. As *commit* is in-order, the first core to *commit* is that with the lowest *specID:* 1. Now, a new thread can execute speculatively on core 1, with a *specID* of N+1. Repeating this process till the time when core N is about to commit, core N-1 has a *specID* of N+N-1. Now, after core N commits and a new thread wishes to execute speculatively on it, N can be subtracted from each *specID* so that each core i now has again been assigned with a *specID* of i.

In the presence of squashes or when the number of speculative threads is lesser than the number of cores, the maximum *specID* is clearly lesser than $2N$-1. Therefore, it is sufficient to keep the size of each entry in the *isSp* vector to be $lg(2N\text{-}1)$ and thus, the storage overhead of the *isSp* vector is $Nlg(2N\text{-}1)$.

The analysis of design complexity due to *MSHR* and cache controller is similar to that of the L1 cache case, discussed in Subsection [3.2.1].

3.4 Other Design Considerations

Designing a cache subsystem and its possible interactions with the core and hence the runtime naturally gives scope to a lot of design space exploration. Some important attributes that have had the most impact on the design are discussed below.

3.4.1 Multiple Writers

In theory, because commits are executed in order, speculative writes to the same word need not be coherent *w.r.t.* all speculative cores as long as they can be read only from their own core. A scheme was initially developed to support this by adding an *isM* bit vector to the directory for each block, in a manner similar to the *isShared* bit vector. This, however, causes the bookkeeping overhead in the directory to double. This approach has been discarded because of the notions that making a directory scalable is still a topic of current research [22, 23], and that writes are not generally on the critical path.

An approach to allowing multiple writes onto distinct words within the same block is given in [1]. An advantage of this is that it helps avoids false conflicts. This, however, has not been implemented in this design.

3.4.2 Eviction and Context Switch

The simplistic approach taken by this design is to simply *squash* in case of an eviction. Assuming that the entire speculative context of a thread can be contained within an L1 cache, then, subject to associativity, this *squash* is unlikely to occur in the absence of context switches. This is a direct implication of LRU whereby the non-TLS blocks (accessed before core went into TLS mode) will get replaced first. Thus, to prevent capacity related *squash*, it would be useful if the compiler can estimate a thread's context size before marking it as TLS.

Upon context switch or speculative context overflow, it is likely that a TLS

block gets evicted. If, however, a *squash* is to be avoided at this point, then a table storing all TLS block addresses and their state bits of the currently executing context (even after eviction from L1) has to be maintained in order to prevent memory corruption (*i.e.,* to facilitate *rollback*). Storage for this extra bookkeeping could be implemented alongside L1, L2 or as an independent fully associative buffer. This added complexity would further increase the overhead for a context switch or a *rollback*.

3.4.3 Inclusion

As the intent of this design is to simply demonstrate TLS, adding shadow tags in the L2 cache and further complicating the cache coherence protocol to accommodate non-inclusive or exclusive caches was not attempted.

However, if coupled with allowing L1 eviction, using a non-inclusive or exclusive setup would allow running TLS on threads with much bigger speculative contexts.

3.4.4 On Demand Commit

Commit is the process by which (in the absence of squash) the TLS bit in every L1 cache block is turned off, *i.e.,* the following transitions take place: $SpE > E$, $SpS > S$, $SpM > M$. Stalling the processor till all these transitions take place is wasteful. In particular, the last transition: $SpM > M$ requires sending of a message from L1 to the directory, which then sends an invalidation message to all (earlier) sharers of that block and a count to this L1. This L1 has to then wait for all the (inv)acknowledgements from the sharers.

Instead, a simple optimization has been implemented so that the processor can continue to execute (in non-TLS mode post issuing a *commit* request) without stalling, and a *commit* transition is triggered upon a TLS block only when it has been accessed.

Chapter 4

Gem5 Implementation

This chapter gives the details of implementing the design outlined in Chapter [3] into the *gem5* simulator. Sections [4.1], [4.2] and [4.3] describe the role of various files within *gem5* that need to modified, Section [4.4] provides some insight as to how TLS could be utilized from user space code.

Though the TLS design is largely agnostic to the exact *ISA* architecture, an x86 framework is used here simply because of the facts that it is well handled by *gem5* in the presence of a detailed memory system (*Ruby*), and that it is ubiquitously used.

4.1 X86 Core

As explained in Section [3.1], two bits, *viz. TLS* and *SQ*, are required to be implemented within each core. While read access is sufficient for the *SQ* bit, software requires write access to the *TLS* bit. Consider the control register *CR0* [3]:

31	30	29	28	27	...	18	17	16	...	5	4	3	2	1	0
PG	CD	NW	-*	-*	-	AM	-	WP	-	NE	ET	TS	EM	MP	PE

,

where,

```
PG - Paging
CD - Cache Disable
NW - Not Write-through
28 - TLS
27 - SQ
AM - Alignment Mask
```

```
WP - Write Protect
NE - Numeric Error
ET - Extension Type
TS - Task Switched
EM - Emulation
MP - Monitor Co-processor
PE - Protection Enable
```

Manipulation of the originally defined bits is possible with the help of kernel mode privileges. However, direct usage (write, especially) of the reserved bits in this register is not recommended by Intel[3], as is also demonstrated in Appendix [B] where approaches of using new system calls or loadable kernel modules for this purpose fail in an actual environment created with the help of a binary translation based emulator known as *QEMU* [20].

These two bits could be stored in a separate general purpose register, but this would increase register pressure. Or, they could be bit-packed along with the remainder of the program data, but this would make register allocation more complex than it already is, especially in light of context saving and rollback. Both these approaches thus hamper performance.

A more elegant and efficient alternative is to continue to store *TLS* and *SQ* bits in *CR0*, but with an interface through the *ISA* to read and manipulate them. There are 16 bits of unused opcode space in the *ISA,* some of which are utilized by *gem5* itself for implementing simulator specific commands such as *exit(), checkpoint(), panic()* etc..

```
-- src/arch/x86/isa/decoder/two_byte_opcodes.isa
```

- Declare two new instructions 'spstart' and 'spcommit' in place of m5reserved instructions

```
-- src/sim/pseudo_inst.(hh/cc)
```

- Define spstart() function to set CR0.tls and reset CR0.sq
- Define spcommit() function to reset CR0.tls and return CR0.sq

X86 by default has a few operating (sub)modes like real mode, compatibility mode, virtual8086 mode, protected mode and sixtyfourbit mode. Most of these modes are for backward compatibility purposes all the way to 8086 and its limited memory. Upon boot, it is the protected mode that is in use today. Thus, to make the processor 'switch to TLS execution', the protected mode has been duplicated to create a TLS mode.

`-- src/arch/x86/types.hh`

- Add a TLS mode to the list of (sub)modes that includes protected mode

`-- src/arch/x86/isa/decoder/one_byte_opcodes.isa`

- Mimic functionality of protected mode into TLS mode

Just as how the *CR0*.0 (PE) bit is used to enable protected mode, the *CR0*.28 bit is used to enable TLS mode. Details of sequence of steps involved in using TLS correctly is mentioned in Section [4.4].

`-- src/arch/x86/regs/misc.h`

- Define TLS and SQ bits as bit-fields 28 and 27 of CR0

`-- src/arch/x86/isa.cc`

- Tell gem5 that setting CR0.tls implies TLS mode

`-- src/arch/x86/process.cc`

- While setting the initial state to protected mode, reset CR0.tls and CR0.sq

4.2 Interfacing the Core and the Ruby Memory System

As discussed in Section [3.2], the *TLS* and *SQ* bits have to be propagated as part of every memory request. For this purpose, two new attributes are defined.

`-- src/mem/request.hh`

- Define two new attributes (or, flags) called TLS and SquashedTLS

`-- src/cpu/simple/timing.(hh/cc)`

- Before sending an Ifetch or handling a read/write packet, assert appropriate attributes in the request based on the values of CR0.tls and CR0.sq

It is also necessary for the memory subsystem to communicate back to the core regarding a squash. Two new address agnostic attributes are needed to indicate to the core when a squash occurs, and also, when invalidation as a result of squash is complete.

In *gem5*, the CPU and the Ruby memory system communicate by means of the master-slave port mechanism (*cf.* Figure [4.1]). The modules and ports themselves are implemented with liberal use of inheritance and virtual functions. For example, when a *RubyPort* sends a squash request to the processor, it does so by calling the *sendSquashRequest()* function under the scope of *SlavePort*. This gets routed through the base *Port* to the *MasterPort* - which in this case is the *TimingCPUPort* - where a *recvSquashRequest()* is triggered.

`-- src/mem/packet.(hh/cc)`

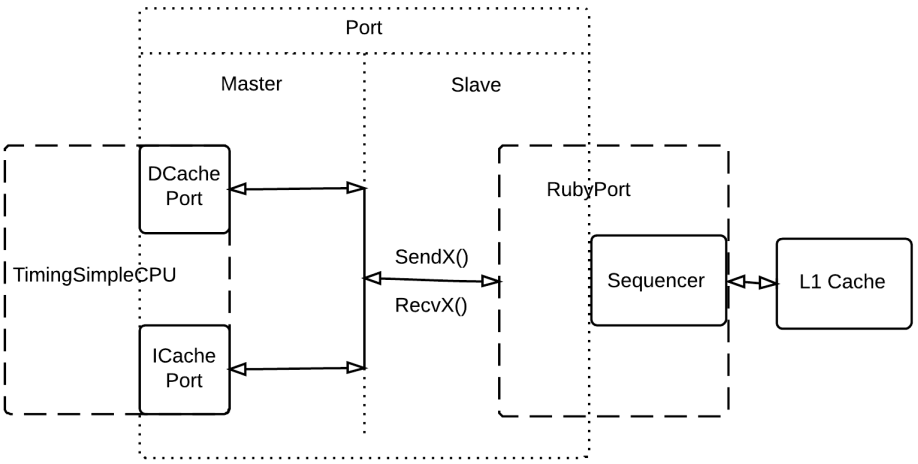


Figure 4.1: Stripped port schematic of the CPU - Ruby Memory interface

- Define two new attributes called SquashReq and ResetSquashReq to indicate a squash detection and completion of invalidation respectively.

`-- src/mem/port.(hh/cc)`

- Implement SlavePort::sendSquashRequest() and SlavePort::sendResetSquashRequest(by triggering the corresponding MasterPort::recv() functions.

- Declare these MasterPort::recv() functions as virtual

`-- src/cpu/simple/timing.(hh/cc)`

- Implement the virtual TimingCPUPort::recv() functions to (re)set CR0.sq upon receipt of (Reset)SquashReq from the memory subsystem

4.3 Ruby Memory System

Each L1 Cache is associated with its own *Sequencer*, which is derived from a *RubyPort*. It is the Sequencer that does the decoding of the attributes of the incoming request from the processor.

-- src/mem/ruby/system/Sequencer.(hh/cc)

- If the TLS attribute is set in the incoming request, then treat every Load as an SpLoad and every Store as an SpStore.

- Else if the SquashedTLS attribute is set, then form a new packet that has to be sent to squash (SpSquash) every speculative address of the associated L1 cache. Upon completion, trigger a RubyResetSquash callback.

- Propagate a Squash callback from the associated L1 cache by triggering a RubySquash callback.

-- src/mem/ruby/system/RubyPort.(hh/cc)

- Implement the Ruby(Reset)Squash callbacks by creating a new (Reset)SquashReq packet and passing it to SlavePort::send(Reset)SquashRequest()

-- src/mem/protocol/RubySlicc_Exports.sm

- Declare requests such as SpLoad, SpStore and SpSquash

Once a request is issued by the Sequencer, it is picked up by the associated L1 cache. The request is then processed as per the cache coherency protocol implementation, which is elaborated in detail in Appendix [A].

The relevant files are:

src/mem/protocol/TLS_MESI_CMP_directory-L1Cache.sm
src/mem/protocol/TLS_MESI_CMP_directory-L2Cache.sm
src/mem/protocol/TLS_MESI_CMP_directory-dir.sm
src/mem/protocol/TLS_MESI_CMP_directory-msg.sm
src/mem/protocol/TLS_MESI_CMP_directory-dma.sm

`src/mem/protocol/TLS_MESI_CMP_directory.slicc`

`src/mem/protocol/SConsopts`

`configs/ruby/TLS_MESI_CMP_directory.py`

The implementation of the coherency protocol is done in a DSL (Domain Specific Language) called *SLICC*. However, this is more suited for handling requests at a per-block level. *I.e.,* implementing the necessary *isSp* and *specID* framework (*cf.* Section [3.3]) turns out to be rather round-about if implemented in *SLICC*. Such cache-level bookkeeping is thus done at the base C++ level itself.

`-- src/mem/ruby/system/CacheMemory.(hh/cc)`

- Use an std::map<NodeID, int> mapping for the isSp vector
- Add methods to add/remove a new TLS node, query specID of a node, get all younger specIDs, get all younger speculative sharers etc.

`-- src/mem/protocol/RubySlicc_Types.sm`

- Declare the methods defined in CacheMemory for compatibility with SLICC.
- These methods are invoked by the cache coherence protocol (TLS_.*\.sm files) to help trigger the appropriate transition and action

4.4 End User

In order to access the instructions *spstart* and *spcommit*, these must be exposed from *gem5* via an includable header file and a linkable assembly file.

`-- util/m5/m5op.h`

- Declare the two functions that are callable from user-space code

`-- util/m5/m5ops.h`

- Map opcode bits to the functions declared in the two_byte_opcodes.isa file

`-- util/m5/m5op_x86.S`

- Define the callable functions by inserting the mapped opcode bits into the assembly code

To summarize the utility of these instructions, Figure. [4.2] shows the high level actions that occur during the execution of each TLS thread. It is expected that appropriate source transformation is done so that this is mapped onto a parallel runtime such as pthread, and that appropriate rollback or sequential fall-back is effected in case a squash is detected. These are explained in more detail in Section [5.1].

As is indicated by the meaning of the two bits *TLS* and *SQ* (*cf.* Table [3.1]), two sequences are possible for the pair:

- Successful Commit: (00) -> (10) -> (00)
- Squashed Commit : (00) -> (10) -> (11) -> (01) -> (00)

This implementation was tested using the Ruby Random Tester that comes with Gem5, and with a few micro-benchmarks written specifically to test the functionality of the intended design. Analysis based on these micro-benchmarks is done in Chapter [5].

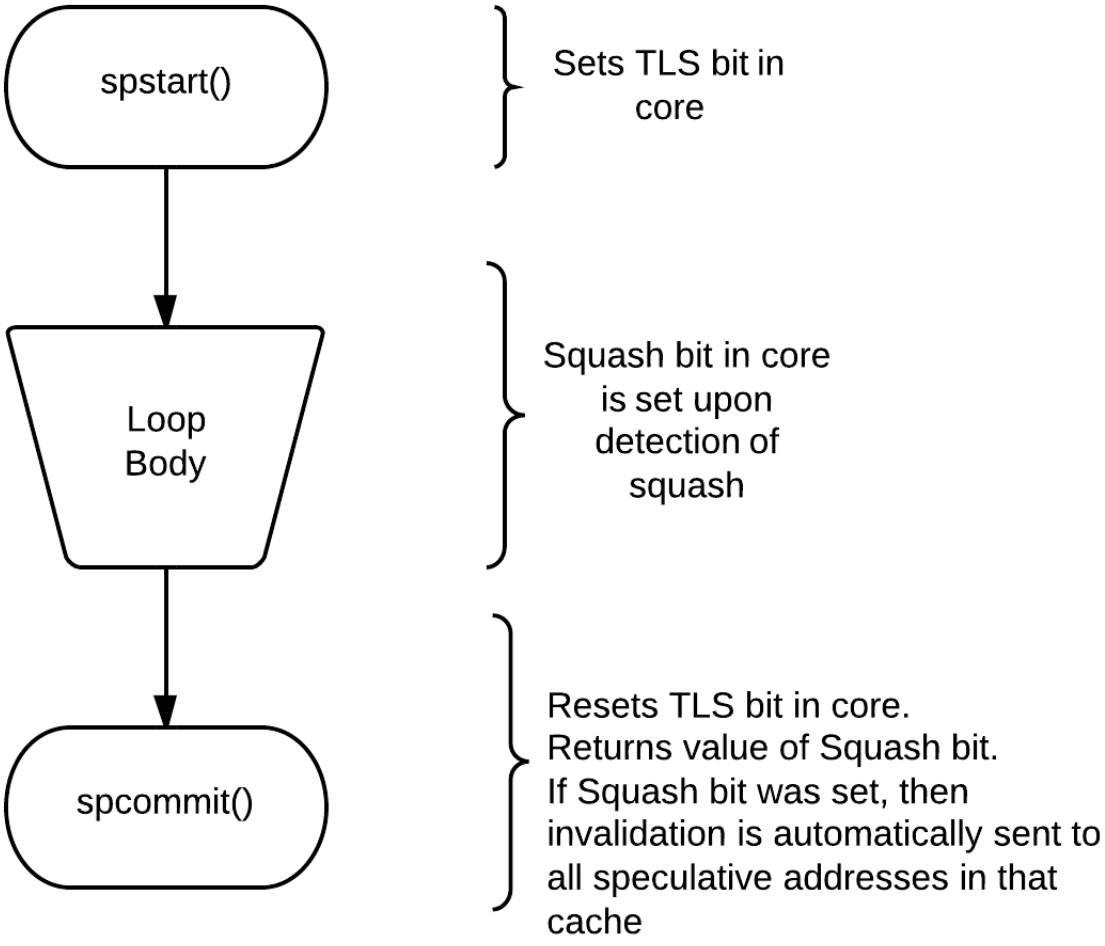


Figure 4.2: Execution of a single TLS thread

Chapter 5

Analysis

This Chapter presents an analysis of performance of the implemented TLS framework in *gem5* from the perspective of the end user. For this purpose, statically compiled *C* binaries were run on *gem5* in its SE (Syscall Emulation) mode, with a fixed set of microarchitectural parameters (*cf.* Table [5.1]). These values have in no way been biased towards obtaining speedup for TLS specifically. However, performance may vary upon changing these parameters. For example, it is intuitive that reducing the cache size increases the probability of a capacity miss, and hence, the probability of a *squash* (smaller maximum speculative context), which hurts performance (*cf.* Subsection [3.4.2]).

Section [5.1] details the parallelizing environment used to leverage TLS, and also outlines two static approaches to handle a *squash*. Section [5.2] describes the behaviour of the system when subject to a few common memory patterns. These Sections strive to give an insight as to where TLS can be beneficial. Likewise, Section [5.3] presents some performance details of running a few *SPEC* 2006 benchmarks [17] on a commercial TLS implementation - BlueGene Q [13]. These

Parameter	Value
CPU Type	Timing - gem5 specific
Memory System Type	Ruby - gem5 specific
ISA	X86
CPU Speed	2 GHz
Memory Size	512 MB
L1 I Size	32 KB
L1 D Size	64 KB
L1 Associativity	2
Cache Line Size	64 B
L2 Size	2 MB
L2 Associativity	8

Table 5.1: Fixed microarchitectural parameters in *gem5*

benchmarks, however, were not run on the *gem5* implementation because of a lack of compiler support to render the source code into a compatible TLS binary.

5.1 Pthread Environment

The *pthread* paradigm was chosen in preference over others such as *OpenMP* and *Cilk* simply for the reason that the former provides much more flexibility and control than the latter two. Moreover, *gem5* supports a light-weight implementation of most of the commonly used functions of *pthread*s, called *m5_threads.* All that is necessary is to link the *pthread* user code with *m5_threads* before passing it on to *gem5.*

As can be seen in Figures [5.1] and [5.2], the role of the *pthread* environment is as follows:

- Wrap the loop body to be parallelized into a function.
- Insert TLS calls into the loop body.
- Manage spawning and joining of threads subject to the following:
  - In-order spawn to facilitate correct detection of squash. This is required because the architecture assigns *specID*s on a first come first serve basis, *i.e.,* for the squash detection mechanism to work correctly, it is essential that an *earlier/older* speculative thread is assigned a lower *specID* than a *later/younger* thread.
  - In-order commit to preserve program semantics in the case where a *squash* is detected.

When it comes to preserving program semantics in the case of a *squash,* in a static environment, there are two approaches that serve as the extremes of possibilities, *viz.* basic and aggressive. The former, as depicted in Figure [5.1], resorts to sequential execution from the latest successful commit prior to the earliest detected *squash.* The latter, as depicted in Figure [5.2], performs a rollback and retry in case a *squash* is detected. Though the latter approach may succeed in extracting more parallelism, if present, from the loop, it comes at the cost of having to save the speculative context before the start of each TLS thread, thereby

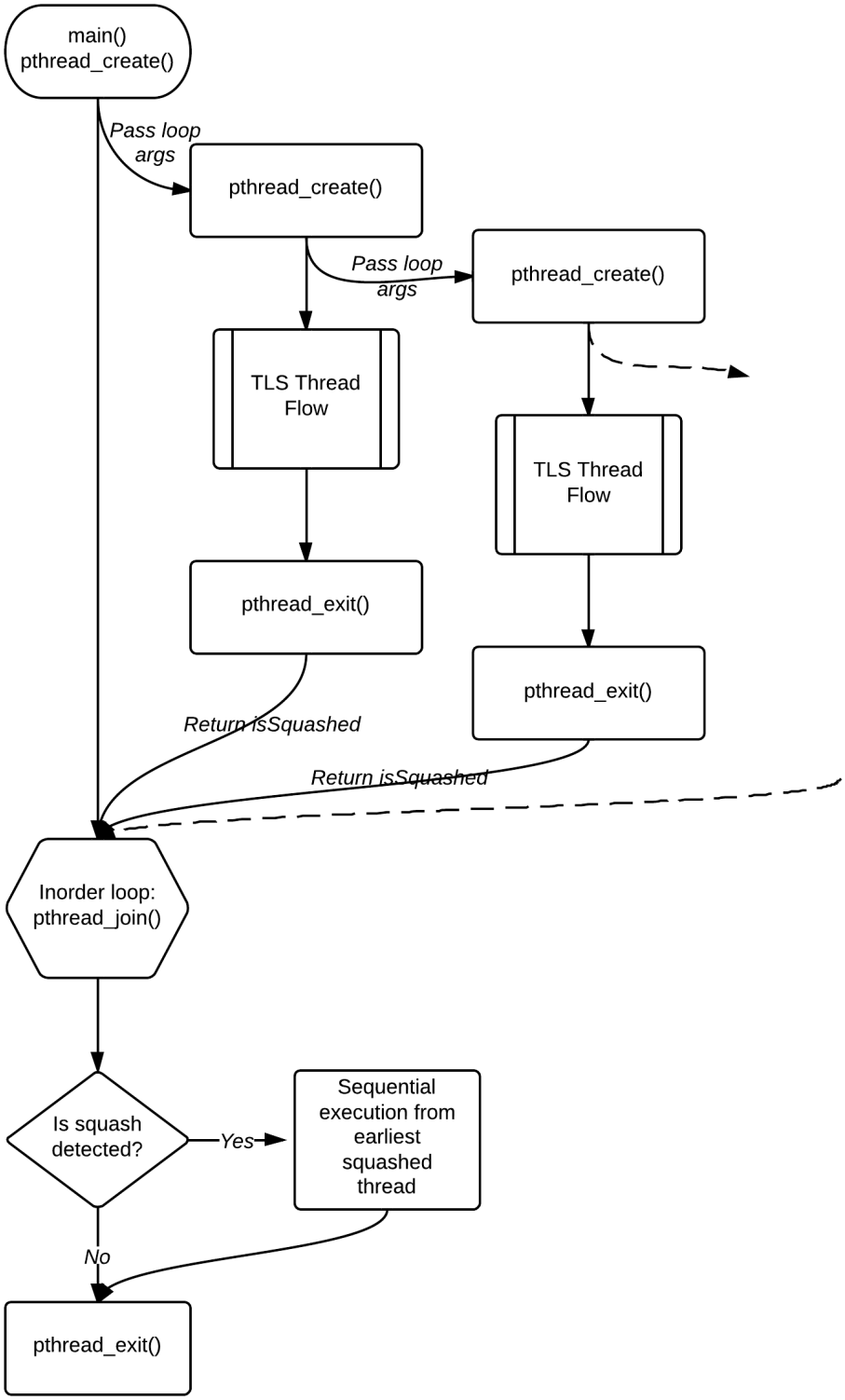


Figure 5.1: A conservative approach of sequential fallback in case of squash. The TLS Thread Flow process is shown in Figure [4.2].

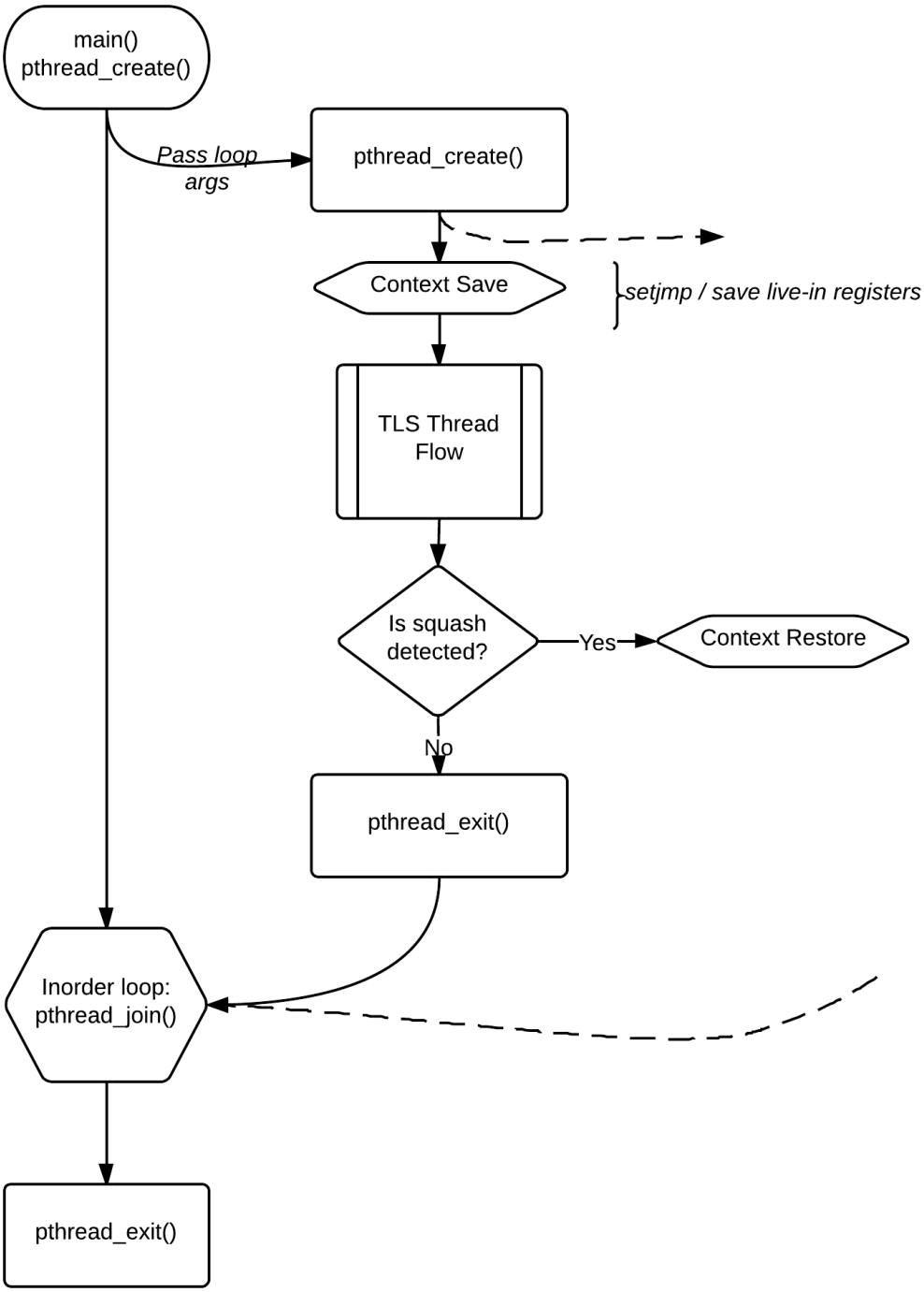


Figure 5.2: An aggressive approach of rollback and retry in case of squash. The TLS Thread Flow process is shown in Figure [4.2].

causing an overhead even in the case of execution of a perfectly parallelizable loop. This is quantified in Subsection [5.2.1].

5.2 Micro-benchmarks

A few micro-benchmarks were written in *C* to test out the functionality of the intended design and to demonstrate the utility of TLS. These memory access patterns were then manually transformed (*cf.* Section [5.1]) to incorporate *pthreads* and TLS, and were statically compiled and linked with *m5_threads* for execution with *gem5*. *gettimeofday* was used to measure the execution time of the Region Of Interest (ROI). For sequential code, the ROI is nothing but the loop itself, but for parallelized code, the ROI also includes the thread creation and join sections. For each of these micro-benchmarks, three independent runs were conducted: the sequential version (*serial*), the parallel version (*pthread*) and the parallel version complete with calls and control flow to utilize TLS (*pthread+TLS*). A separate printing of the output was done outside the critical section as a means of manually verifying program semantics for all the three versions.

Also included are inherently sequential paradigms where TLS would obviously *squash*. This is simply to re-iterate the fact it is not prudent for the compiler or the end-user to over-optimistically parallelize without first doing atleast a preliminary analysis of the source code.

5.2.1 Array Modification

Here, modification of several memory locations is attempted in parallel. However, as is the case when pointers or variable inputs are involved, parallelizing compilers refuse to extract any parallelism from such loops (irrespective of whether the iterations are actually independent or not) simply because of the presence of inter-iteration *may* dependence. With TLS, this could be overcome.

As each core has its own private L1 cache, parallelism can be attempted at the granularity of a cache line. Consider the cases where the array modifications

occur to different cache lines, to different words in the same cache line, and to the same word. It is to be noted that the cache line size has been fixed at 64 bytes, and that the size of *int* is 4 bytes.

Different cache lines

```
for (i = 0; i < N; i++) {
  int k;
  for (k = 0; k < LOOP_BODY_SIZE; k++) {
    a[16*i+N-1] = k;
  }
}
```

It is not feasible to perform a *must* dependence analysis on this code, especially if the value of N is dependent upon the input. However, when N is less than 16, it is clear (to a human or to a very intelligent compiler) that the memory writes would occur at distinct cache lines. This means that this code does possess parallelism, which can be exploited by TLS.

Each version of this code, *viz.* *serial*, *pthread* and *pthread+TLS*, was run independently with a wide dynamic range of $LOOP\_BODY\_SIZE$. For the TLS version, both strategies to handle *squash* (*cf.* Section[5.1]) were evaluated. Figures [5.3] and [5.4] show the variation of speedup of the parallel versions of the code over the serial version with different $LOOP\_BODY\_SIZE$, when the basic sequential fallback and the aggressive rollback and retry approaches are used for the TLS version respectively. Figure [5.5] shows the profile of the speedup (slowdown) that TLS overhead causes when compared to a bare-bones *pthread* version, again, with both the *squash* handling strategies.

As can be seen, it is not worth parallelizing loops that have a small loop body. One possible approach would hence be to unroll the loop appropriately before setting it up for parallelization. It may appear that TLS causes significant overhead in the case of rollback, but this is a worst-case unoptimized software situation. The source of the overhead is actually the *setjmp* framework used

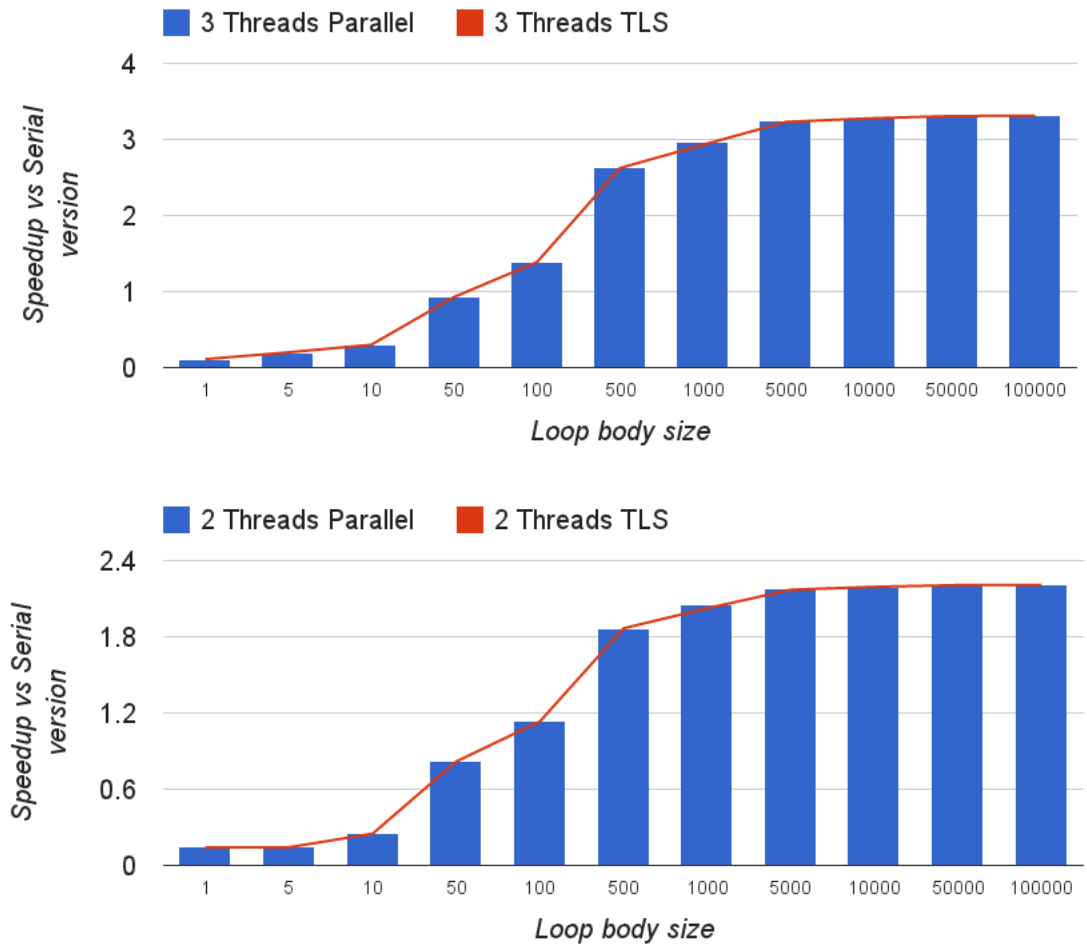


Figure 5.3: Speedup of parallel codes when compared with the serial version, without the use of setjmp

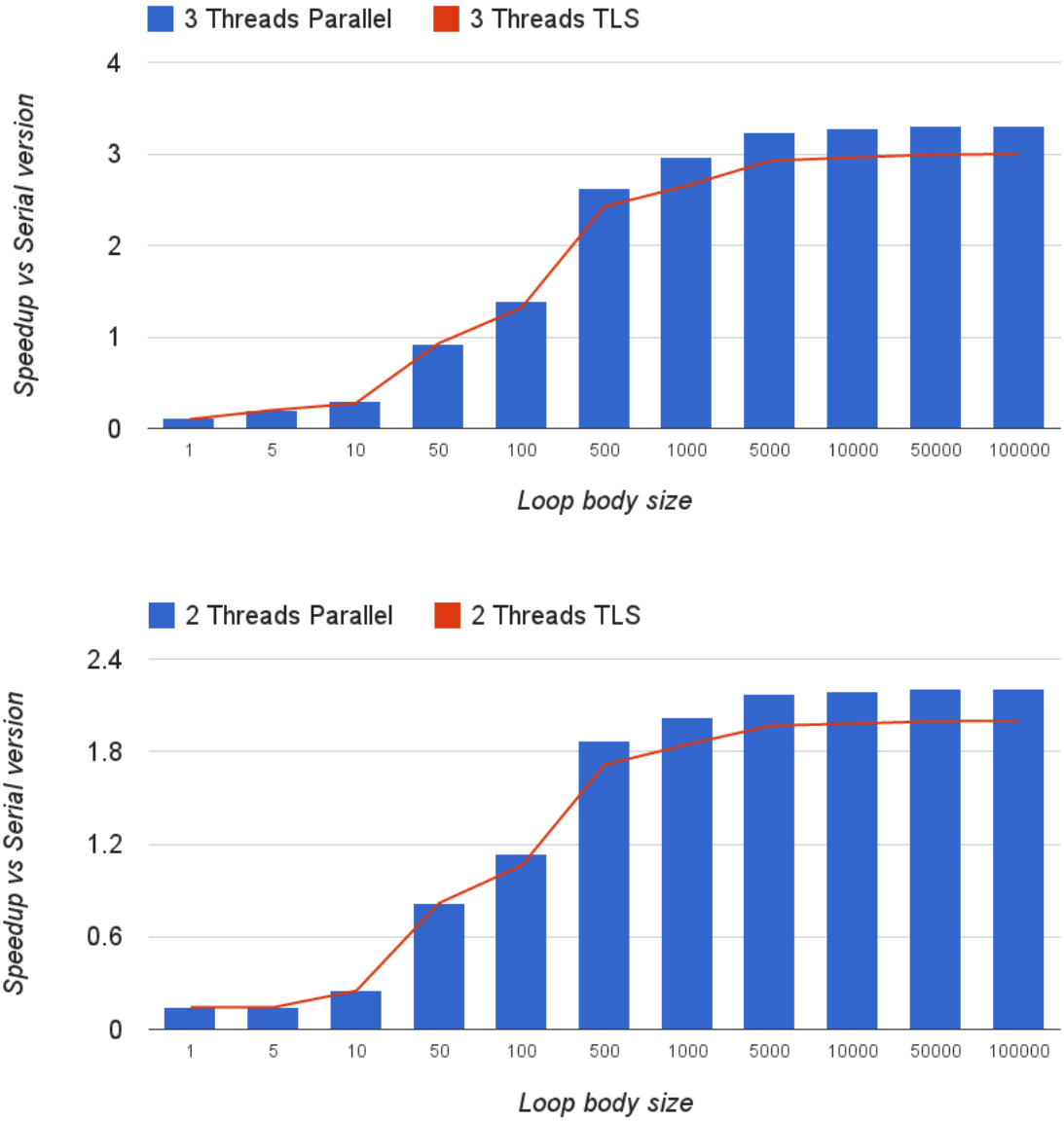


Figure 5.4: Speedup of parallel codes when compared with the serial version, with the use of setjmp

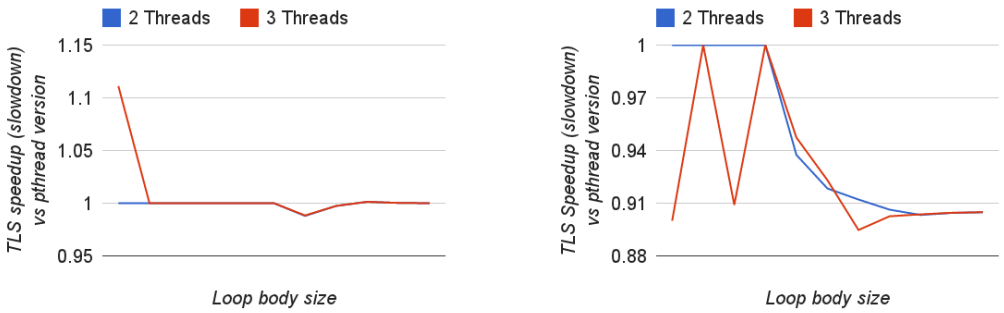


Figure 5.5: TLS overhead over pthread version - with (right) and without (left) out the use of setjmp

to store a context for possible restoration later. *Setjmp* ends up saving all the registers as part of the context, and this can typically include registers that are not live-in at that program point. As this is wasteful, it would be worthwhile for the compiler to explicitly save only the live-in registers, if it analyses that the loop is not perfectly independent. The figure to the left in Figure [5.5] clarifies this position that TLS in itself causes negligible overhead in the parallelizing scenario.

As mentioned earlier in this use-case, it must be noted that an auto-parallelizing compiler would typically not schedule such a loop for parallelization because of the presence of *may* dependence and that the value of N may be dependent upon the input. Therefore, the gain in performance when using TLS may very well be assessed based on the *serial* version in this use-case.

The geometric means of these numbers are available in Table [5.2].

Different words - same cache line

```
for (i = 0; i < N; i++) {
  int k;
  for (k = 0; k < LOOP_BODY_SIZE; k++) {
    a[i] = k;
  }
}
```

Here, as the same cache line is being modified, using TLS results in a *squash* for every thread after the first. Even if a sequential fallback is done after the second thread, the overall code is observed to be slower than the sequential version because of the overhead involved in *pthread*.

However, when N is greater than 16, if sufficient analysis is done by the compiler, it is possible to unroll this loop, or equivalently, effect the following transformation to then pass to TLS:

```
for (i = 0; i < N; i += 16) {
  int k;
```

```
  for (k = 0; k < LOOP_BODY_SIZE; k++) {
    int j;
    for (j = i; j < i+16; j++) {
      a[j] = k;
    }
  }
}
```

Same word and cache line

```
for (i = 0; i < N; i++) {
  int k;
  for (k = 0; k < LOOP_BODY_SIZE; k++) {
    a[0] = k;
  }
}
```

In this case, no amount of source transformation would render parallelism possible; TLS always shows slowdown because of *squash*.

5.2.2 Array Access

The experimental setup and intent are almost identical as in Subsection [5.2.1]. However, as there is no modification involved, only two use-cases are worth considering: one where parallel reads are attempted on different cache lines, and the other, on the same cache line.

Different Cache Line Access

```
for (i = 0; i < N; i++) {
  int k, x;
  for (k = 0; k < LOOP_BODY_SIZE; k++)
    x = a[16*i+N-1];
```

```
  }
}
```

Same Cache Line Access

```
for (i = 0; i < N; i++) {
  int k, x;
  for (k = 0; k < LOOP_BODY_SIZE; k++)
    x = a[i];
  }
}
```

As the intent of this analysis is primarily to gauge utility of the implemented TLS architecture, the sequential fallback strategy was implemented for these use-cases. The rollback and retry strategy suffers from software overhead because of *setjmp* (*cf.* Subsection [5.2.1]), thus the former strategy is more representative of the performance of the underlying architecture.

As is expected, it can be seen from Figures [5.6] and [5.7] that both these use-cases do return near identical speedup profiles, and that the overhead due to TLS is indeed negligible (Figure [5.8]).

Table [5.2] summarizes the performance data for the array access and array modification use-cases where no *squash* is detected. As $LOOP\_BODY\_SIZE$ can often be determined statically by the compiler, also included is data pertaining to the case where a speedup of atleast 1 is achieved *w.r.t.* the serial version.

5.2.3 Migratory and Producer-Consumer

Migratory

```
a[0] = N;
for (i = 0; i < N; i++) {
  int k;
  for (k = 0; k < LOOP_BODY_SIZE; k++) {
```

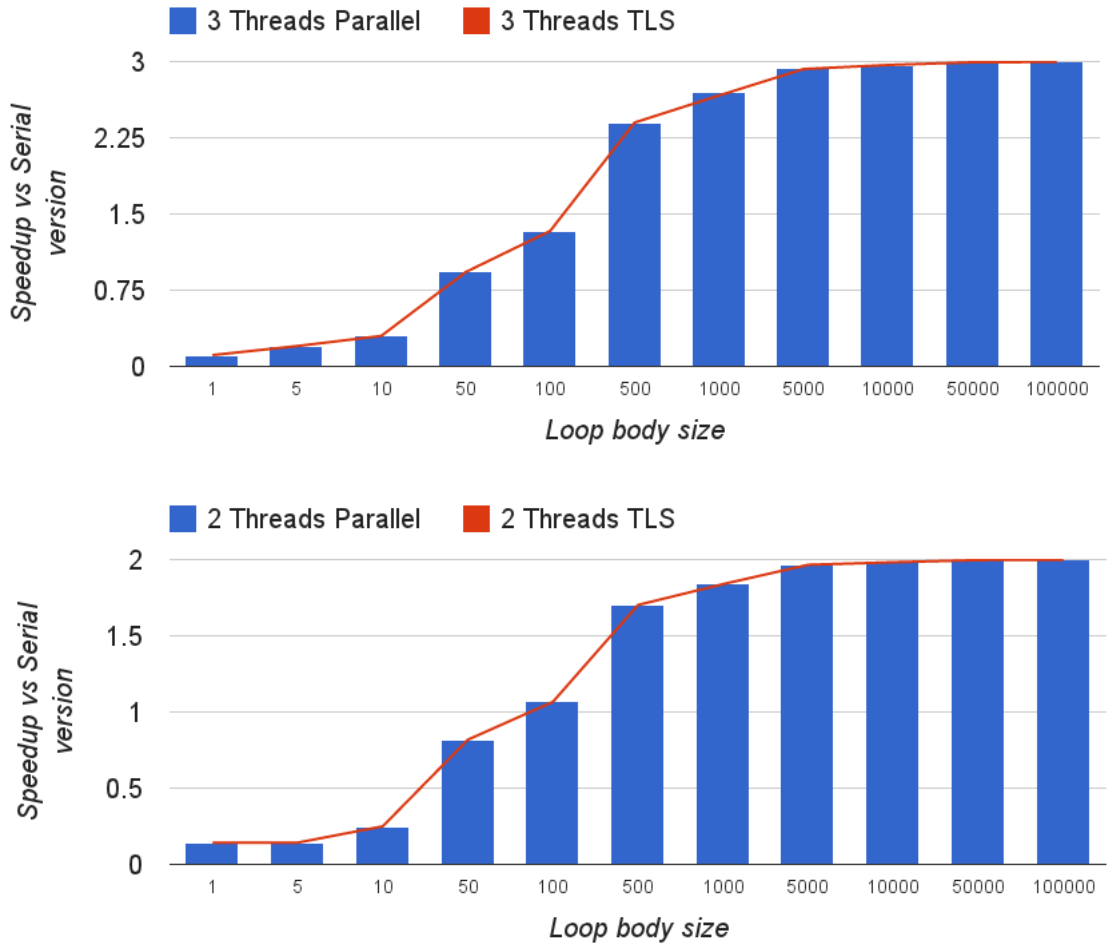


Figure 5.6: Speedup of parallel codes when compared with the serial version, for the different cache line access use case

Use-Case			TLS Speedup (Geometric Mean)			
	# of Cores	Setjmp	LOOP_BODY_SIZE $\geq$ 100		All	
			vs. Serial	vs. Parallel	vs. Serial	vs. Parallel
Write a[16*i+N-1]	2	Y	1.76	0.91	0.87	0.94
		N	1.93	1.00	0.92	1.00
	3	Y	2.53	0.91	1.12	0.93
		N	2.77	1.00	1.20	1.00
Read a[16*i+N-1]	2	N	1.76	1.00	0.87	1.00
	3	N	2.53	1.00	1.14	1.00
Read a[i]	2	N	1.93	1.00	0.92	1.00
	3	N	2.77	1.00	1.20	1.01

Table 5.2: Summary of TLS speedup behaviour in absence of squash

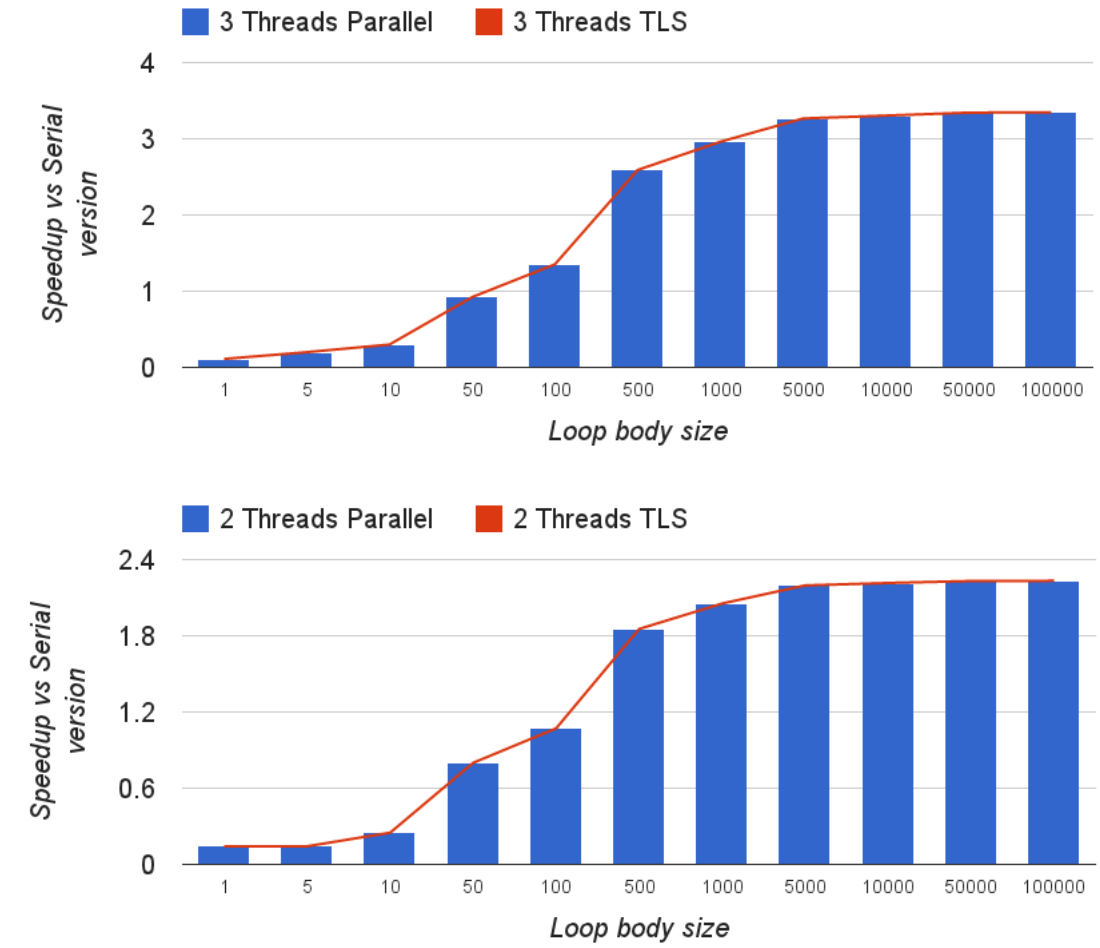


Figure 5.7: Speedup of parallel codes when compared with the serial version, for the same cache line access use case

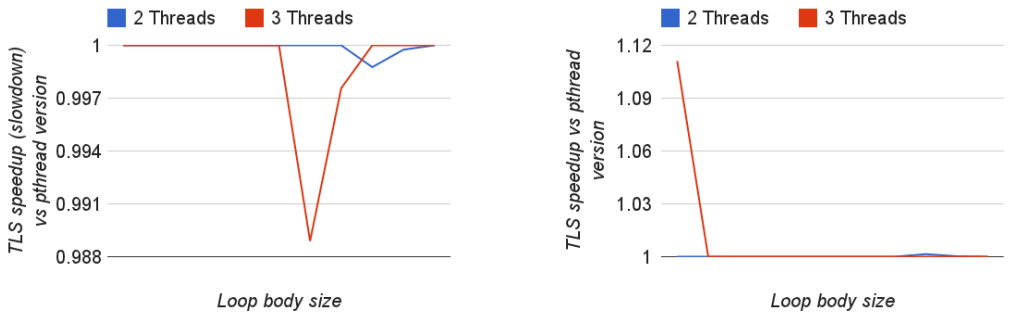


Figure 5.8: TLS overhead over pthread version - for the different cache line access use-case (left) and for the same cache line access use-case (right)

```
      a[16*(i+1) + N-1] = a[0];
    }
  }
```

In this case, the cache line that contains *a[0]* is written to precisely once, and all subsequent accesses to this line are reads. Thus, this particular use case does have parallelism that can be exploited. However, a *squash* is observed because, from the perspective of the architecture, there is no guarantee that the core which wrote to the *a[0]* line does not do so again. In other words, the *a[0]* line is in M state, and remains so unless evicted or invalidated. Thus, no further speculative reads (or writes) are possible to this line unless it gets flushed from the modifying cache. If speedup has to be achieved in this case, compiler/end-user support is necessary. A comparable scenario occurs in the case of the producer consumer. Moreover, if the stride width is indeterminable statically, profiling becomes necessary.

Producer-Consumer

```
for (i = 0; i < N; i++) {
  int k;
  for (k = 0; k < LOOP_BODY_SIZE; k++) {
    if (i == N-1) {
      b[16*i + N-1] = a[16*(i-(N-1)) + N-1];
    } else {
      b[16*i + N-1] = 1;
    }
    a[16*i + N-1] = i + N-1;
  }
}
```

As can be seen from the behaviour of the implementation, it is worth re-stating that the TLS architecture only guarantees correctness but not speedup. It is the role of the end-user - together with the compiler - to ensure that squashes are kept to a minimum. This is not singular to this implementation, but holds true for

TLS architectures in general, as, for instance, is described in Section [5.3].

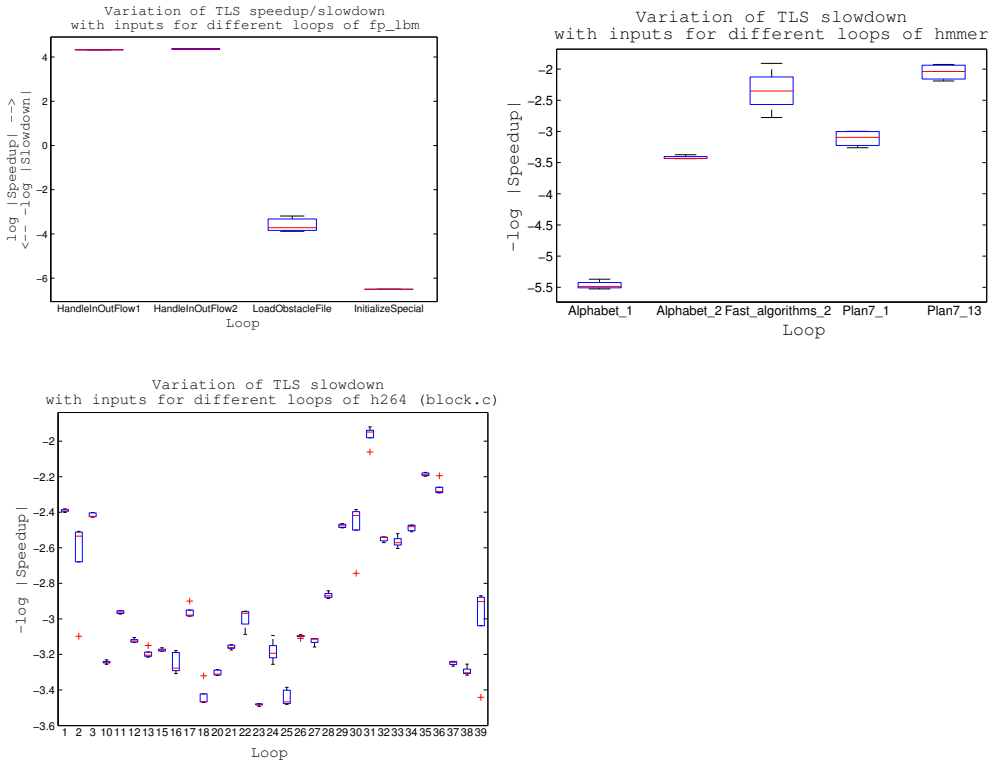
5.3 BlueGene Q

As mentioned earlier, due to a lack of compiler support to render code into a TLS binary that is compatible with the implementation presented in this work, analysis of only micro-benchmarks was done. However, to provide further insight into the nature of TLS execution so that it may be utilized by the compiler/end-user more efficiently, this Section presents analysis based on a few experiments done on the *BlueGene*Q system.

This was actually done as part of another related work [16] but is being reproduced here for cohesion.

5.3.1 Input Dependent Speedup

To determine variation in TLS speedup/slowdown of a given program when subject to different inputs, experiments were done on three *SPEC* 2006 benchmarks, viz. *fp_lbm*, *hmmer* and *h264*. A *bash* script was used to insert TLS or OpenMP *pragma* calls to some of the loops and to insert calls to measure number of cycles using *bgpm*, which is a C interface to access hardware performance measurement counters on BlueGene. A compiler that supports these pragma calls on BG/Q, *bgxlc_r,* was used. Speedup comparable to *openMP* speedup was observed *w.r.t. fp_lbm* but slowdowns were observed for *hmmer* and *h264*. Each candidate loop was executed speculatively and non-speculatively in separate runs and this was done for every input. The cycles measured during every such execution are for the execution of the corresponding loop alone.

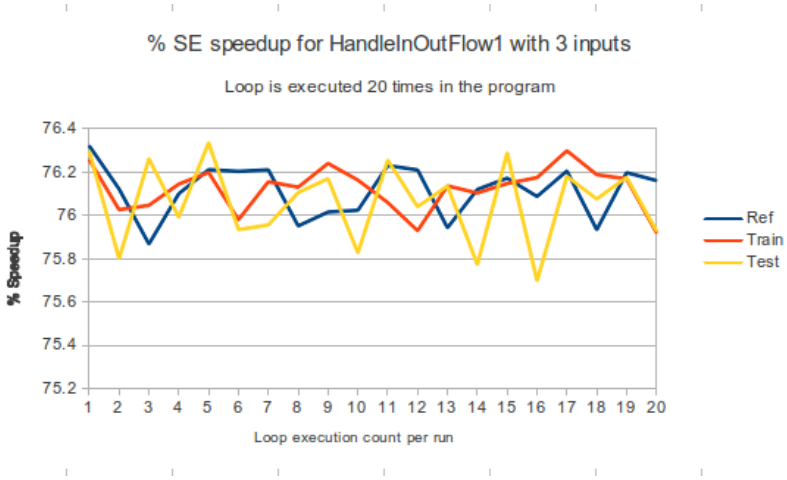


As the speedup varies significantly depending upon the loop, a log box-plot of the absolute value of speedup is shown so that effect of input on speedup (or slowdown) can be seen. The inputs used for *fp_ lbm* were *ref*, *train* and *test*; those for *hmmer* were *bombesin*, *leng100*, *nph3* and *retro;* and those for *h264* were *foreman_ ref_ encoder_ baseline*, *foreman_ ref_ encoder_ main*, *foreman_ test_ encoder_ baseline*, *foreman_ train_ encoder_ baseline* and *sss_ encoder_ main*.

While speedup is largely independent of inputs for *fp_ lbm*, this is not the case for *hmmer* and *h264*. (It must be noted that a logarithmic scale was used for the box-plots to accommodate the high dynamic range of the values.)

5.3.2 Speedup across multiple invocations of a loop

It is possible that extent of TLS speedup/slowdown of a loop depends not only on the input but also on the calling context. To measure this, each loop invocation was executed speculatively and non-speculatively in separate runs and this was done for every input. However, for *fp_ lbm*, it was observed for two of the loops that speedup was largely independent of context and input.


% SE speedup for HandleInOutFlow1 with 3 inputs
Loop is executed 20 times in the program
% Speedup
76.4
76.2
76
75.8
75.6
75.4
75.2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Loop execution count per run
Ref
Train
Test

Chapter 6

Conclusion

As is the case with any optimization or design that hopes to *improve* a certain metric of code execution, this work satisfies the three main requirements:

1. **Correctness**. The main attraction of TLS lies in the fact that it guarantees correctness of execution by detecting conflicts at runtime. This is done by embedding the *specID* framework into the cache coherence protocol, which ensures that the directory automatically issues a *squash* to all the cores that have made use of stale data. (*cf.* Chapter [3] and Appendix [A]). In Section [5.2], all relevant memory locations were printed out to manually verify correctness for all the micro-benchmarks that were written.

2. Demonstrated (*cf.* Section [5.2]) improvement in **performance** and its scope. The term *improved performance,* in this context, has two components:
   (a) *TLS is used only in the cases where auto-parallelizers fail to parallelize a loop.* There are several scenarios - involving pointers and input-dependent parameters - where auto-parallelizing compilers fail to extract parallelism. It has been demonstrated in such cases that the use of TLS results in significant speedup when compared to the otherwise sequential execution. Even if the code is written to explicitly parallelize loops in such cases, there is no significant difference in performance between this explicit parallel version and the TLS version.
   (b) *TLS is used as a replacement for an existing parallelizing framework.* It has been demonstrated that the underlying architectural implementation, by itself, results in an identical performance profile when used with or without TLS, when used on codes for which it is semantically correct to parallelize without TLS.

   However, the scope of achieving speedup cannot be generalized to all programs, as it is meaningless to try and parallelize programs that are inherently sequential in nature.

3. Ease of **adoptability**. The bare-bones interface for software to interact with the processor to leverage TLS has been designed and implemented in a very simplistic manner (*cf.* Section [4.4]). However, as mis-speculation hampers performance significantly depending upon how often a *squash* occurs and how the *squash*-handling is done, a greater challenge lies in providing compiler support that provides a guaranteed speedup (or atleast, one that guarantees that there is no slowdown) for any given code.

As it is very clear that compiler support is a must to properly utilize TLS, the following is a brief summary of the insights gained through this work regarding program analysis in a TLS environment, and, resulting suggestions for the compiler/end-user to incorporate.

Like every other parallel framework, it is imprudent to attempt parallelization when the the size of the loop body is *small*. The notion of *small* is obviously dependent on the entire system, but this can be easily calibrated; either independently or by means of a profiling run. With the framework implemented in this work, it is recommended that a minimum loop body size of ≈100 load/stores be used (*cf.* Section [5.2]). On the other hand, care must be taken that the speculative context size doesn't exceed the L1 capacity. This is because, as described in Subsection [3.4.2], there is no provision to store the speculative state of evicted blocks, and hence, a speculative context overflow would cause performance degradation due to *squash*. As extracting these parameters from source code is often possible statically (infrastructure to do this in LLVM [21] - using the *opt* tool - has been implemented as part of another related work [16]), loop unrolling can be used to arrive at an optimal value.

However, it is certainly not sufficient to halt analysis once this optimal size has been reached. The compiler must also be cache line aware, *i.e.,* take into account the granularity of cache accesses to minimize false conflicts, for instance, in the second use-case in Subsection [5.2.1].

If the *squash*-handling mechanism is some version of rollback and retry, another dimension gets added to this trade-off in deciding as to *how much* of a loop has to be speculated upon at a time. To elaborate, when a rollback/retry is involved, it becomes necessary to save and restore context. As it is wasteful to simply save and restore all registers as part of the context, the compiler must involve only the *live-in* registers at the program point and that too only when it analyzes that a *squash* is probable. For instance, upon saving all the registers using *setjmp*, a performance drop of ≈10% was observed (*cf.* Subsection [5.2.1]) when no such saving was actually required. Thus, if a *squash* is probable, another way of partitioning the loop for TLS is by minimizing the number of *live-in* registers in the resulting

critical section.

Further, if it is possible for the compiler to detect a strided dependency, it is necessary for it to vary the TLS spawn width it issues so as to minimize the probability of a *squash.* However, performing such analyses statically can be unreliable as well as expensive. For example, in the migratory/producer-consumer use-cases as shown in Subsection [5.2.3], it is impossible to determine stride in the presence of input-dependent parameters. Coupled with the fact that speedup is dependent on the loop and input, as evidenced by Section [5.3], and in some cases, possibly dependent on the execution context as well, it becomes necessary to incorporate dynamic profiling support to complete this cost-benefit analysis. To give a simple illustration, assuming input dependence and context independence - as was found in Subsections [5.3.1] and [5.3.2], the following basic profiling may be done: If there are multiple calls to a loop, execute the loop sequentially the first time, speculatively the second time and measure the speedup; if speedup is observed, execute the remaining invocations of the loop speculatively, else, fallback to sequential execution. This way, it is less likely that TLS is attempted on a loop where no speedup can be achieved. Having mentioned that, it must be noted that program profiles vary significantly from the sample of the SPEC 2006 and the micro-benchmarks that were used in this work.

However, one of the prime utilities of TLS is to enable compilers to *optimistically* parallelize. Spending a lot of time on analyses of code defeats the purpose as it may still turn out that a given loop was simply not fit to be parallelized in any manner. As it may be obvious to the end-user if a given loop is inherently sequential, it becomes useful to provide a framework of annotations so that hints can be given to the compiler.

There is also scope for further improving this *gem5* implementation. For example, another design choice is to use separate (physical/virtual) links to service messages that are needed only to satisfy speculative requests, as, it is undesirable that non-TLS requests get starved because of all the resources (bandwidth, buffers etc.) being used up by TLS requests alone. The utility of this design choice may become more apparent when a more detailed scalability analysis is done.

As an example of an application specific optimization, to reduce false conflicts in the migratory use-case (Subsection [5.2.3]), cache (line) *flush* can be effected if the architecture can get a hint that a given line is going to be written precisely once. Or, more simply, traffic can be traded for *squashes* by employing data forwarding from an older speculative core that writes to a line to a newer one.

Moreover, as Transactional Memory (TM) is based on a similar design principle of speculation, this work may be extended to support it. Lastly, bringing some dynamism to the runtime would allow for much more efficient *squash* handling mechanisms than the static approaches that were discussed in Section [5.1]. For example, instead of retrying or simply giving up on parallelism, synchronization primitives can be built upon this framework.

In a nutshell, several possibilities exist in this design space that involves compile time, compiler complexity, ease of programming, architecture (design, implementation and verification) complexity, and run time. Ideally, though TLS provides rollback support to ensure correctness, this should be used as sparingly as possible.

Bibliography

[1] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. 2005. The STAMPede approach to thread-level speculation. ACM Trans. Comput. Syst. 23, 3 (August 2005), 253-300.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2 (August 2011), 1-7.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals (Link as of May 2014).

[4] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In Proceedings of the 22nd annual international symposium on Computer architecture (ISCA '95). ACM, New York, NY, USA, 414-425.

[5] Franklin, M.; Sohi, G.S., "ARB: a hardware mechanism for dynamic reordering of memory references," Computers, IEEE Transactions on , vol.45, no.5, pp.552,571, May 1996.

[6] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. 1998. Speculative Versioning Cache. In Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA '98). IEEE Computer Society, Washington, DC, USA, 195-.

[7] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In Proceedings

of the 27th annual international symposium on Computer architecture (ISCA '00). ACM, New York, NY, USA, 1-12.

[8] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. SIGOPS Oper. Syst. Rev. 32, 5 (October 1998), 58-69.

[9] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk Disambiguation of Speculative Threads in Multiprocessors. SIGARCH Comput. Archit. News 34, 2 (May 2006), 227-238.

[10] Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In Proceedings of the 16th International Symposium on Parallel and Distributed Processing (IPDPS '02). IEEE Computer Society, Washington, DC, USA, 20-.

[11] Marcelo Cintra and Diego R. Llanos. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03). ACM, New York, NY, USA, 13-24.

[12] Cosmin E. Oancea and Alan Mycroft. 2008. Software thread-level speculation: an optimistic library implementation. In Proceedings of the 1st international workshop on Multicore software engineering (IWMSE '08). ACM, New York, NY, USA, 23-32.

[13] Haring, R.A.; Ohmacht, M.; Fox, T.W.; Gschwind, M.K.; Satterfield, D.L.; Sugavanam, K.; Coteus, P.W.; Heidelberger, P.; Blumrich, M.A.; Wisniewski, R.W.; Gara, A.; Chiu, G.L.-T.; Boyle, P.A.; Chist, N.H.; Changhoan Kim, "The IBM Blue Gene/Q Compute Chip," Micro, IEEE , vol.32, no.2, pp.48,60, March-April 2012.

[14] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. 2012. Evaluation of Blue Gene/Q hardware support for transactional memories. In Pro-

ceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12). ACM, New York, NY, USA, 127-136.

[15] Arnamoy Bhattacharyya. Do inputs matter? Using data-dependence profiling to evaluate thread level speculation in the BlueGene/Q. MSc Thesis, University of Alberta, 2013.

[16] Sriseshan Srikanth. Automatic parallelization using TLS on BG/Q. MITACS Globalink research internship, University of ALberta, 2013. http://www.ee.iitm.ac.in/~ee09b060/pdf/mitacsReport.pdf (Link as of May 2014).

[17] SPEC2006 Benchmark suite. http://www.spec.org/cpu2006/ (Link as of May 2014).

[18] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," January 2005. http://sesc.sourceforge.net (Link as of May 2014).

[19] http://gem5.org/Special:RecentChanges (Link as of May 2014).

[20] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, Berkeley, CA, USA, 41-41.

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04). IEEE Computer Society, Washington, DC, USA, 75-

[22] Daniel Sanchez and Christos Kozyrakis. 2012. SCD: A scalable coherence directory with flexible sharer set encoding. In Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12). IEEE Computer Society, Washington, DC, USA, 1-12.

[23] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. 2011. Cuckoo directory: A scalable directory for many-core systems. In Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11). IEEE Computer Society, Washington, DC, USA, 169-180.

Appendix A

Cache Coherence Protocol State Machine

This detailed state machine has been written in a manner that makes it easy to implement in *gem5*, which in turn uses a DSL (Domain Specific Language) called *SLICC*. As the emphasis is on debug-ability, there is some redundancy *w.r.t.* some transient states and messages.

To make the terminology used clearer, an example trace of the protocol with a two core system is explained in the following table.

	L1 Cache - 0	L1 Cache - 1	L2 Cache
0	*State*: M	*State*: I	*State*: M
		Event: Store	
		Action: Allocate a Data Block (Trigger an L1 Replacement if needed) Allocate a TBE (Similar to MSHR) Issue a GetM message to L2 Pop the request	
		Next State: I_M^{AD}	
1	*State*: M	*State*: I_M^{AD}	*State:* M
			Event: GetM
			Action: Forward request to Owner Make requester Owner Mark requester as not Speculative Set block as most recently used Pop the request
			Next State: M
2	*State:* M	*State*: I_M^{AD}	*State:* M
	Event: Fwd_GetM		
	Action: Send Data to Requester Deallocate L1 Block Pop the request		
	Next State: I		
3	*State:* I	*State*: I_M^{AD}	*State:* M
		Event: Data_Core	
		Action: Write Data to L1 Block	
		Indicate Store Hit	
		Deallocate TBE	
		Pop the request	
		Wakeup Dependants	
		(Process stalled events)	
		Next State: M	

The coherence messages used in this implementation are mentioned in the next two tables.

Message	Description of L1 Request
GetS	Get Shared
GetM	Get Modified
GetInstr	Get Instruction
GetSpS	Get Speculatively Shared
GetSpM	Get Speculatively Modified
UpdateAsSq	Indicate that Block has been Invalidated post Squash
ToM	Upgrading to M
ToSpM	Going to SpM
ToSpMData	Going to SpM with Writeback Data
ToSp	Going to SpS/SpE
FromSp	Commit from SpS/SpE/SpM
EtoI	Indicate that E Block wishes to be Invalidated
MtoI	Send Writeback Data to indicate that M Block wishes to be invalidated
StoI	Indicate that S Block wishes to be Invalidated
Fwd_GetM	Forward of an unserviced forwarded GetM request
SquashReq	Indicate that core is not speculative anymore
DummyFwd	Forward of an unserviced forwarded dummy request

Message	Description
Data_Excl	Exclusive Data from L2
Data_Ack	Data with an appended Ack count from L2
Data_NoAck	Data indicating no more Acks are necessary from L2
Data_Core	Inter-cache transfer of Data
NumAcks	Ack count from L2
Ack	Ack from L2
Squash	Squash from L2
Inv	Invalidate from L2
EtoS	Downgrade from L2
InvAck	Invalidation Ack from L1
Mem_Data	Data from Memory
Mem_Ack	Ack from Memory
Mem_Inv	Invalidate from Memory

A.1 L1 Cache

Having mentioned the meaning of the 7 base states in Table [3.2], the transient states are described in the following table.

State	Meaning
I_S^D	Issued GetS, waiting for data
$I\_S^D\_I$	Need to invalidate after wait in I_S^D is satisfied
I_M^{AD}	Issued GetM, waiting for data and ack(s)
I_M^A	Received data, waiting for ack(s)
$I\_M^A\_S$	Need to share after wait in I_M^A is satisfied
$I\_M^A\_I$	Need to invalidate after wait in I_M^A is satisfied
$I\_M^A\_S\_I$	Need to invalidate after wait in $I\_M^A\_S$ is satisfied
I_SpS^D	Issued GetSpS, waiting for data
I_SpM^D	Issued GetSpM, waiting for data
E_I^A	Sent EtoI, waiting for Ack
M_I^A	Sent MtoI, waiting for Ack
S_I^A	Sent StoI, waiting for Ack
SpM_MAa	Issued FromSp, waiting for all younger sharers to invalidate
S_MAa	Issued ToM, waiting for all (non-speculative) sharers to invalidate
$S\_M^A\_S$	Need to share after wait in S_M^A is satisfied
$S\_M^A\_I$	Need to invalidate after wait in S_M^A is satisfied
$S\_M^A\_S\_I$	Need to invalidate after wait in $S\_M^A\_S$ is satisfied
S_SpM^A	Issued ToSpM, waiting for ack
S_SpE^A	Issued ToSp, waiting for ack
S_SpS^A	Issued ToSp, waiting for ack

A.1.1 Processor Triggered Events

Table 1.1a	Load / Ifetch	SpLoad	Store	SpStore
I	Alloc. D/I Block Alloc. TBE GetS, Pop	Alloc. D Block Alloc. TBE, GetSpS Set isSp, Pop	Alloc. D Block Alloc. TBE GetM, Pop	Alloc. D Block Alloc. TBE, GetSpM Set isSp, Pop
	I_S^D	I_SpS^D	I_M^{AD}	I_SpM^D
S	Load Hit Pop	ToSp, Alloc. TBE Set isSp, Pop	Alloc. TBE ToM, Pop	ToSpM, Alloc. TBE Set isSp, Pop
	-	S_SpS^A	S_M^{Aa}	S_SpM^A

Table 1.1a	Load / Ifetch	SpLoad	Store	SpStore
E	Load Hit Pop	ToSp, Alloc. TBE Set isSp, Pop	Alloc. TBE ToM, Pop	ToSpM, Alloc. TBE Set isSp, Pop
	-	S_SpE^A	S_M^{Aa}	S_SpM^A
M	Load Hit Pop	ToSpM with Data Load Hit, Set isSp Pop	Store Hit Pop	ToSpM with Data Store Hit, Set isSp Pop
	-	SpM	-	SpM
SpS	Load Hit FromSp Reset isSp Pop	Load Hit Pop	FromSp, ToM Reset isSp Alloc. TBE Pop	ToSpM Alloc. TBE Pop
	S	-	S_M^{Aa}	S_SpM^A
SpE	Load Hit FromSp Reset isSp Pop	Load Hit Pop	FromSp, ToM Reset isSp Alloc. TBE Pop	ToSpM Alloc. TBE Pop
	E	-	S_M^{Aa}	S_SpM^A
SpM	Load Hit FromSp Reset isSp Alloc. TBE Pop	Load Hit Pop	Store Hit FromSp Reset isSp Alloc. TBE Pop	Store Hit Pop
	SpM_M^{Aa}	-	SpM_M^{Aa}	-
I_S^D $I\_S^D\_I$ I_M^{AD} I_M^A $I\_M^A\_S$ $I\_M^A\_I$ $I\_M^A\_S\_I$ I_SpS^D I_SpM^D E_I^A M_I^A SpM_M^{Aa} S_I^A S_M^{Aa} $S\_M^A\_S$ $S\_M^A\_I$ $S\_M^A\_S\_I$ S_SpS^A S_SpM^A	Stall	Stall	Stall	Stall
S_SpE^A	-	-	-	-

Table 1.1b	SpSquash	L1_Replacement	
	assert !isSp		isSp
I	Dummy Store Hit Pop	Ill	Ill
	-	-	-
S	Ill	StoI, Alloc. TBE Dealloc. Block	Ill
	-	S_I^A	-
E	Ill	EtoI, Alloc. TBE Dealloc. Block	Ill
	-	E_I^A	-
M	Ill	MtoI with Data Alloc. TBE	Ill
	-	M_I^A	-
SpS	Dummy Store Hit UpdateAsSq	StoI, Alloc. TBE Dealloc. Block	Squash to core and Dir Reset isSp

Table 1.1b	SpSquash	L1_Replacement	
	assert !isSp		isSp
	Dealloc. Block, Pop		Dealloc. Block
	I	S_I^A	I
SpE	Dummy Store Hit UpdateAsSq Dealloc. Block, Pop	EtoI, Alloc. TBE Dealloc. Block	Squash to core and Dir Reset isSp Dealloc. Block
	I	E_I^A	I
SpM	Dummy Store Hit UpdateAsSq Dealloc. Block, Pop	MtoI with Data Alloc. TBE	Squash to core and Dir Reset isSp Dealloc. Block
	I	M_I^A	I
I_SpS^D I_SpM^D S_SpS^A S_SpM^A S_SpE^A	Dummy Store Hit UpdateAsSq Dealloc. TBE Dealloc. Block, Pop Wake Dependants	Stall	Ill
	I	-	-
I_S^D $I\_S^D\_I$ I_M^{AD} I_M^A $I\_M^A\_S$ $I\_M^A\_I$ $I\_M^A\_S\_I$ E_I^A M_I^A SpM_M^{Aa} S_I^A S_M^{Aa} $S\_M^A\_S$ $S\_M^A\_I$	Ill	Stall	Ill
$S\_M^A\_S\_I$	-	-	-

A.1.2 Non-Processor Triggered Events

Table 1.2a	Data_Excl	Data_Ack	Data_NoAck	Data_Core
I_S^D	Update L1, Load Hit Dealloc. TBE Pop Wake Dependants	Ill	Update L1, Load Hit Dealloc. TBE Pop Wake Dependants	Update L1, Load Hit Dealloc. TBE Pop Wake Dependants
	E	-	S	S
$I\_S^D\_I$	Update L1, Load Hit Dealloc. TBE, L1 Pop Wake Dependants	Ill	Update L1, Load Hit Dealloc. TBE, L1 Pop Wake Dependants	Update L1, Load Hit Dealloc. TBE, L1 Pop Wake Dependants
	I	-	I	I
I_M^{AD}	Ill	Update L1, Record pending Acks in TBE, Pop	Update L1, Store Hit Dealloc. TBE Pop Wake Dependants	Update L1, Store Hit Dealloc. TBE Pop Wake Dependants
	-	I_M^A	M	M
I_SpS^D	Update L1, Load Hit Dealloc. TBE Pop Wake Dependants	Ill	Update L1, Load Hit Dealloc. TBE Pop Wake Dependants	Update L1, Load Hit Dealloc. TBE Pop Wake Dependants
	SpE	-	SpS	SpS
I_SpM^D	Ill	Ill	Update L1, Store Hit Dealloc. TBE Pop Wake Dependants	Update L1, Store Hit Dealloc. TBE Pop Wake Dependants

Table 1.2a	Data_Excl	Data_Ack	Data_NoAck	Data_Core
	-	-	*SpM*	*SpM*
S_M^{Aa}	*Ill*	*Ill*	*Ill*	Update L1, Store Hit Dealloc. TBE Pop Wake Dependants
	-	-	-	*M*
$S\_M^A\_I$	*Ill*	*Ill*	*Ill*	Update L1, Store Hit MtoI Pop
	-	-	-	M_I^A
I *S* *E* *M* *SpS* *SpE* *SpM* I_M^A $I\_M^A\_S$ $I\_M^A\_I$ $I\_M^A\_S\_I$ E_I^A M_I^A SpM_M^{Aa} S_I^A $S\_M^A\_S$ $S\_M^A\_S\_I$ S_SpS^A S_SpM^A	*Ill*	*Ill*	*Ill*	*Ill*
S_SpE^A	-	-	-	-

Table 1.2b	NumAcks	Ack	Squash	EtoS
I	*Ill*	Pop	*Ill*	*Ill*
	-	-	-	-
S	*Ill*	*Ill*	Squash to Core Dealloc. L1 Reset isSp, Pop	*Ill*
	-	-	*I*	-
E	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	Pop
	-	-	-	*S*
M	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
SpS	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
SpE	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	Pop
	-	-	-	*SpS*
SpM	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
I_S^D	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
$I\_S^D\_I$	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
I_M^{AD}	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
I_M^A	*Ill*	*Ill*	Squash to Core	*Ill*

Table 1.2b	NumAcks	Ack	Squash	EtoS
			Reset isSp, Pop	
	-	-	-	-
$I\_M^A\_S$	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
$I\_M^A\_I$	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
$I\_M^A\_S\_I$	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
I_SpS^D	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
I_SpM^D	*Ill*	*Ill*	Squash to Core Reset isSp, Pop	*Ill*
	-	-	-	-
E_I^A	*Ill*	Dealloc. TBE Pop Wake Dependants	Squash to Core Reset isSp Pop	Pop
	-	*I*	-	-
M_I^A	*Ill*	Dealloc. TBE, L1 Pop Wake Dependants	Squash to Core Reset isSp Pop	*Ill*
	-	*I*	-	-
SpM_M^{Aa}	Record pending Acks in TBE, Pop	Dealloc. TBE Pop Wake Dependants	Squash to Core Reset isSp Pop	*Ill*
	-	*M*	-	-
S_I^A	*Ill*	Dealloc. TBE Pop Wake Dependants	Squash to Core Reset isSp Pop	*Ill*
	-	*I*	-	-
S_M^{Aa}	Record pending Acks in TBE, Pop	Store Hit Dealloc. TBE, Pop Wake Dependants	Squash to Core Reset isSp Pop	*Ill*
	-	*M*	-	-
$S\_M^A\_S$	Record pending Acks in TBE, Pop	Store Hit, Data to Recorded and L2, Dealloc. TBE and L1, Pop Wake Dependants	Squash to Core Reset isSp Pop	*Ill*
	-	*S*	-	-
$S\_M^A\_I$	Record pending Acks in TBE, Pop	Store Hit Data to Recorded MtoI with data Pop	Squash to Core Reset isSp Pop	*Ill*
	-	M_I^A	-	-
$S\_M^A\_S\_I$	Record pending Acks in TBE, Pop	Store Hit, Data to Recorded and L2, Dealloc. TBE and L1, Pop Wake Dependants	Squash to Core Reset isSp Pop	*Ill*
	-	*I*	-	-
S_SpS^A	*Ill*	Load Hit Dealloc TBE, Pop Wake Dependants	Squash to Core Reset isSp Pop	Fwd back to L2 Pop
	-	*SpS*	-	-
S_SpM^A	*Ill*	Store Hit Dealloc TBE, Pop Wake Dependants	Squash to Core Reset isSp Pop	Fwd back to L2 Pop
	-	*SpM*	-	-
S_SpE^A	*Ill*	Load Hit Dealloc TBE, Pop	Squash to Core Reset isSp	Fwd back to L2

Table 1.2b	NumAcks	Ack	Squash	EtoS
		Wake Dependants	Pop	Pop
	-	*SpE*	-	-

Table 1.2c	Dummy_Fwd	Fwd_GetS/Fwd_GetInstr	Fwd_GetM
I	Fwd back to L2 Pop	Fwd back to L2 Pop	Fwd back to L2 Pop
	-	-	-
S	Data to Req Pop	Data to Req Pop	*Ill*
	-	-	-
E	*Ill*	*Ill*	*Ill*
	-	-	-
M	Data to Req Pop	Data to Req, L2 Pop	Data to Req Dealloc. L1, Pop
	-	*S*	*I*
SpS	*Ill*	*Ill*	*Ill*
	-	-	-
SpE	*Ill*	*Ill*	*Ill*
	-	-	-
SpM	Data to Req Reset IsSp Pop	Data to Req, L2 Reset IsSp Pop	Data to Req Reset IsSp Dealloc. L1, Pop
	-	*S*	*I*
I_S^D	Fwd back to L2 Pop	Fwd back to L2 Pop	Fwd back to L2 Pop
	-	-	-
$I\_S^D\_I$	*Ill*	*Ill*	*Ill*
	-	-	-
I_M^{AD}	Stall	Stall	Stall
	-	-	-
I_M^A	Stall	Stall	Record Req Pop
	-	$I\_M^A\_S$	$I\_M^A\_I$
$I\_M^A\_S$	Stall	Stall	Record Req Pop
	-	-	$I\_M^A\_S\_I$
$I\_M^A\_I$	*Ill*	*Ill*	*Ill*
	-	-	-
$I\_M^A\_S\_I$	Stall	Stall	*Ill*
	-	-	-
I_SpS^D	*Ill*	*Ill*	*Ill*
	-	-	-
I_SpM^D	*Ill*	*Ill*	*Ill*
	-	-	-
E_I^A	*Ill*	Pop	Pop
	-	-	-
M_I^A	Data to Req Pop	Data to Req Pop	Data to Req Pop
	-	-	-
SpM_M^{Aa}	*Ill*	*Ill*	*Ill*
	-	-	-
S_I^A	Data to Req Pop	Data to Req Pop	*Ill*
	-	-	-
S_M^{Aa}	Stall	Stall	Record Req Pop
	-	$S\_M^A\_S$	$S\_M^A\_I$
$S\_M^A\_S$	Stall	Stall	Record Req Pop
	-	-	$S\_M^A\_S\_I$
$S\_M^A\_I$	Stall	Stall	Stall
	-	-	-

Table 1.2c	Dummy_Fwd	Fwd_GetS/Fwd_GetInstr	Fwd_GetM
$S\_M^A\_S\_I$	Stall	Stall	*Ill*
	-	-	-
S_SpS^A	Fwd back to L2 Pop	Fwd back to L2 Pop	Fwd back to L2 Pop
	-	-	-
S_SpM^A	Fwd back to L2 Pop	Fwd back to L2 Pop	Fwd back to L2 Pop
	-	-	-
S_SpE^A	Fwd back to L2 Pop	Fwd back to L2 Pop	Fwd back to L2 Pop
	-	-	-

Table 1.2d	Inv		InvAck		
	From L2/L1	From L1		Last Ack	<-else
I	InvAck to Req Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
S	InvAck to Req Dealloc. L1, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	I	-	-	-	-
E	InvAck to Req Dealloc. L1, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	I	-	-	-	-
M	MtoI with Data Alloc. TBE, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	M_I^A	-	-	-	-
SpS	InvAck to Req Reset isSp Dealloc. L1, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	I	-	-	-	-
SpE	InvAck to Req Reset isSp Dealloc. L1, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	I	-	-	-	-
SpM	MtoI with Data Reset isSp Alloc. TBE, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	M_I^A	-	-	-	-
I_S^D	InvAck to Req	*Ill*	*Ill*	*Ill*	*Ill*
	$I\_S^D\_I$	-	-	-	-
$I\_S^D\_I$	Stall	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
I_M^{AD}	Stall	InvAck to Req Pop	ack– Pop	*Ill*	*Ill*
	-	-	-	-	-
I_M^A	Stall	*Ill*	*Ill*	ack– Dealloc. TBE Store Hit, Pop Wake Dependants	ack– Pop
	-	-	-	*M*	-
$I\_M^A\_S$	Stall	*Ill*	*Ill*	ack– Dealloc. TBE Store Hit Data to L2, Pop Wake Dependants	ack– Pop
	-	-	-	*S*	-
$I\_M^A\_I$	Stall	*Ill*	*Ill*	ack– Store Hit Data to Recorded MtoI with Data Pop	ack– Pop
	-	-	-	M_I^A	-

Table 1.2d	Inv		InvAck		
	From L2/L1	From L1		Last Ack	<-else
$I\_M^A\_S\_I$	Stall	*Ill*	*Ill*	ack– Store Hit Data to Recorded, L2 Dealloc. TBE, L1 Pop Wake Dependants	ack– Pop
	-	-	-	*I*	-
I_SpS^D	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
I_SpM^D	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
E_I^A	Stall	InvAck to Req Pop	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
M_I^A	Stall	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
SpM_M^{Aa}	*Ill*	*Ill*	*Ill*	ack– Dealloc. TBE, Pop Wake Dependants	ack– Pop
	-	-	-	*M*	-
S_I^A	Stall	InvAck to Req Pop	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
S_M^{Aa}	Stall	InvAck to Req Pop	*Ill*	ack– Dealloc. TBE Store Hit, Pop Wake Dependants	ack– Pop
	-	$S\_M^A\_I$	-	*M*	-
$S\_M^A\_S$	Stall	InvAck to Req Pop	*Ill*	ack– Dealloc. TBE Store Hit Data to L2, Pop Wake Dependants	ack– Pop
	-	$S\_M^A\_I$	-	*S*	-
$S\_M^A\_I$	Stall	*Ill*	*Ill*	ack– Store Hit Data to Recorded MtoI with Data Pop	ack– Pop
	-	-	-	M_I^A	-
$S\_M^A\_S\_I$	Stall	*Ill*	*Ill*	ack– Store Hit Data to Recorded, L2 Dealloc. TBE, L1 Pop Wake Dependants	ack– Pop
	-	-	-	*I*	-
S_SpS^A	InvAck to Req Squash to core Load Hit Dealloc. TBE, L1 Reset isSp, Pop	InvAck to Req Pop	*Ill*	*Ill*	*Ill*
	I	-	-	-	-
S_SpM^A	InvAck to Req Squash to core Store Hit Dealloc. TBE, L1 Reset isSp, Pop	InvAck to Req Pop	*Ill*	*Ill*	*Ill*
	I	-	-	-	-
S_SpE^A	InvAck to Req Squash to core Load Hit Dealloc. TBE, L1	InvAck to Req Pop	*Ill*	*Ill*	*Ill*

Table 1.2d	Inv		InvAck		
	From L2/L1	From L1		Last Ack	<-else
	Reset isSp, Pop				
	I	-	-	-	-

A.2 L2 Cache

Having mentioned the meaning of the 4 base states in Table [3.4], the transient states are described in the following table.

State	Meaning
E_I^{Aa}	May have sent squashes. Waiting for InvAck(s) for Inv(s) sent
E_I^{a}	Sent EtoI to Memory. Waiting for Mem_Ack
S_I^{Aa}	May have sent squashes. Waiting for InvAck(s) for Inv(s) sent. Will send StoI to Memory
mS_I^{Aa}	May have sent squashes. Waiting for InvAck(s) for Inv(s) sent. Will send Data to Memory
S_I^{a}	Sent StoI/Data to Memory. Waiting for Mem_Ack
M_Ida	May have sent squashes. Waiting for Data from Owner
I_S^{d}	Waiting for Mem_Data for Memory Fetch to service GetS/GetSpS/GetInstr yet
I_M^{d}	Waiting for Mem_Data for Memory Fetch to service GetM yet
M_I^{a}	Sent Data to Memory. Waiting for Mem_Ack
M_S^{D}	Received a Sharer request while in M. Waiting for Data.

Table 2a	GetInstr(/GetS)			
		Dir is Owner	<-else	Owner isSp
I	Alloc. Block, TBE Req is Owner Record Req Fetch to Memory (Req isn't Sp) Set MRU, Pop	*Ill*	*Ill*	*Ill*
	I_S^{D}	-	-	-
E	EtoS to Owner Data_NoAck to Req Owner, Req isShared Dir is Owner (Req isn't Sp) Set MRU, Pop	*Ill*	*Ill*	*Ill*
	S	-	-	-
S	Data_NoAck to Req Req isShared (Req isn't Sp) Set MRU, Pop	*Ill*	*Ill*	*Ill*
	-	-	-	-
M	*Ill*	Data_NoAck to Req, Req isShared (Req isn't Sp) Set MRU, Pop	Fwd to Owner Owner, Req isShared Alloc. TBE (Req isn't Sp) Set MRU, Pop	Data_NoAck to Req, Req isShared (Req isn't Sp) Set MRU, Pop
	-	S	M_S^{D}	-
E_I^{Aa} E_I^{a} S_I^{Aa} mS_I^{Aa} S_I^{a}	Stall	*Ill*	*Ill*	*Ill*
M_I^{da}	-	-	-	-

Table 2a	GetInstr(/GetS)			
		Dir is Owner	<-else	Owner isSp
I_S^d	Record Req Owner, Req isShared Dir is Owner (Req isn't Sp) Set MRU, Pop	*Ill*	*Ill*	*Ill*
	-	-	-	-
I_M^d M_I^a	Stall	*Ill*	*Ill*	*Ill*
M_S^D	-	-	-	-

Table 2b	GetSpS				
		Owner isSp, Req > Owner	Owner isSp, Req < Owner	Dir is Owner	Owner not Sp
I	Alloc. L2, TBE Req is Owner Record Req Req isSp Addr is Sp Set MRU, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	I_S^d	-	-	-	-
E	EtoS to Owner Data_NoAck to Req, Owner isShared Req isShared Dir is Owner Req isSp Mark addr as Sp Set MRU, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	S	-	-	-	-
S	Data_NoAck to Req, Req isShared Req isSp Mark addr as Sp Set MRU, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
M	*Ill*	Squash to >= Req, Dummy Fwd to Owner, Pop	Data_NoAck to Req, Req isShared Mark addr as Sp Req isSp Set MRU, Pop	Data_NoAck to Req, Req isShared,isSp Mark addr as Sp Req isSp Set MRU, Pop	Squash to >= Req, Dummy Fwd to Owner, Pop
	-	-	-	*S*	-
E_I^{Aa} E_I^a S_I^{Aa} mS_I^{Aa} S_I^a	Stall	*Ill*	*Ill*	*Ill*	*Ill*
M_I^{da}	-	-	-	-	-
I_S^d	Record Req Owner isShared Dir is Owner Req isShared Req isSp Mark addr as Sp Set MRU, Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
I_M^d M_I^a	Stall	*Ill*	*Ill*	*Ill*	*Ill*
M_S^D	-	-	-	-	-

Table 2c	GetM			
		Dir is Owner / Owner is Req	<-else	Owner isSp
I	Alloc. Block, TBE Req is Owner Record Req Req isn't Sp Fetch to Memory Set MRU, Pop	*Ill*	*Ill*	*Ill*
	I_M^d	-	-	-
E	*Ill*	Data_NoAck to Req Req is Owner Req isn't Sp Set MRU Pop	Data_Ack_1 to Req Inv (Fwd) to Owner Req is Owner Req isn't Sp Set MRU Pop	Data_NoAck to Req Squash to isSp >= Owner, Req is Owner Req isn't Sp Mark addr as not Sp Set MRU, Pop
	-	M	M	M
S	Inv (Fwd) to Non-Sp Sharers, Data_Ack to Req Squash to >= first instance of isShared&isSp, Req isn't Sp Set MRU, Pop	*Ill*	*Ill*	*Ill*
	M	-	-	-
M	*Ill*	Data_NoAck to Req Req is Owner Req isn't Sp Set MRU Pop	Fwd to Owner Req is Owner Req isn't Sp Set MRU Pop	Squash to >= min(Owner, first Sp Sharer), Inv (Fwd) to Non-Sp Sharers, Data_Ack to Req Req is owner Req isn't Sp Set MRU, Pop
	-	-	-	-
E_I^{Aa} E_I^{a} S_I^{Aa} mS_I^{Aa} S_I^{a}	Stall	*Ill*	*Ill*	*Ill*
	-	-	-	-
M_I^{da}	Stall	Data_NoAck to Req Req is Owner Req isn't Sp Pop	*Ill*	*Ill*
	-	-	-	-
I_S^d I_M^d M_I^a M_S^D	Stall	*Ill*	*Ill*	*Ill*
	-	-	-	-

Table 2d	GetSpM					
		Owner isSp, Req < Owner	Dir is Owner	Owner isn't Sp	Req < instance of isShared&isSp	<-else
I	Alloc. L2 Alloc. TBE Req isOwner Record Req Req isSp Addr is Sp Set MRU Pop	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	I_M^d	-	-	-	-	-

Table 2d	GetSpM					
		Owner isSp, Req < Owner	Dir is Owner	Owner isn't Sp	Req < instance of isShared&isSp	<-else
E	Ill	Data_NoAck to Req, Squash to isSp>=Owner, Req isSp Req isOwner Mark addr as Sp, Set MRU Pop	Ill	Data_NoAck to Req, EtoS to Owner, Owner isShared, Req isOwner Mark addr as Sp, Set MRU, Pop	Ill	Ill
	-	M	-	M	-	-
S	Ill	Ill	Ill	Ill	Squash to isSp >= first such instance, Data_NoAck to Req, Req isOwner Req isSp Mark addr as Sp, Set MRU Pop	Data_NoAck to Req, Req is Owner, Req isSp Mark addr as Sp, Set MRU Pop
	-	-		-	M	M
M	Ill	Data_NoAck to Req, Squash to isSp >=min(Owner, first Sp Sharer after Req), Req isOwner Req isSp Mark addr as Sp, Set MRU Pop	Data_NoAck to Req, Req isOwner Req core isSp Mark addr as Sp, Set MRU Pop	Squash to >= Req, Dummy Fwd to Owner, Pop	Ill	Ill
	-	-	-	-	-	-
E_I^{Aa} E_I^{a} S_I^{Aa} mS_I^{Aa} S_I^{a} M_I^{da} I_S^{d} I_M^{d} M_I^{a}	Stall	Ill	Ill	Ill	Ill	Ill
M_S^{D}	-	-	-	-	-	-

Table 2e	FromSp	ToSpMData	ToSpM			ToSp
				Req < instance of isShared&isSp	<-else	
I	Ill	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-	
E	Req isn't Sp Mark addr as not Sp Pop	Ill	Ack to Req Req isSp Mark addr as Sp, Set MRU Pop	Ill	Ill	Ack to Req Req isSp Mark addr as Sp, Set MRU Pop
	-	-	M	-	-	-

Table 2e	FromSp	ToSpMData	ToSpM			ToSp
				Req < instance of isShared&isSp	<-else	
S	Req isn't Sp Pop	*Ill*	*Ill*	Squash to isSp >= first such instance, Ack to Req, Req isn't Shared, Req isOwner Req isSp Mark addr as Sp, Set MRU Pop	Ack to Req, Req is Owner, Req isn't Shared, Req isSp Mark addr as Sp, Set MRU Pop	Ack to Req Req isSp Mark addr as Sp, Set MRU Pop
	-	-	-	*M*	*M*	-
M	Inv (Fwd) to Non-Sp Sharers, NumAcks to Req Req isn't Sp Mark addr as not Sp Pop	Update L2 Req isOwner Req core isSp Mark addr as Sp, Set MRU Pop	Req isSp Mark addr as Sp, Set MRU Pop	*Ill*	*Ill*	*Ill*
	-	-	-	-	-	-
E_I^{Aa}	Req isn't Sp Mark addr as not Sp Pop	*Ill*	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
E_I^{a}	*Ill*	*Ill*	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
S_I^{Aa}	Req isn't Sp Pop	*Ill*	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
mS_I^{Aa}	Req isn't Sp Pop	Pop	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
S_I^{a}	*Ill*	*Ill*	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
M_I^{da}	Pop	Pop	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
I_S^{d}	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-	-
I_M^{d}	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-	-
M_I^{a}	*Ill*	*Ill*	Pop	*Ill*	*Ill*	Pop
	-	-	-	-	-	-
M_S^{D}	Stall	Stall	Stall	*Ill*	*Ill*	Stall
	-	-	-	-	-	-

Table 2f	EtoI		StoI		MtoI		
		Req isn't Owner		Last Ack		Req is Owner	Last Ack
I	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-	-	-
E	Ack to Req Dir is Owner Pop	*Ill*	*Ill*	*Ill*	Update L2 Ack to Req Dir is Owner Pop	*Ill*	*Ill*
	S	-	-	-	*M*	-	-
S	*Ill*	Ack to Req Req isn't Shared, Pop	Ack to Req Req isn't Shared, Pop	*Ill*	Ack to Req Pop	*Ill*	*Ill*

Table 2f	EtoI		StoI		MtoI		
		Req isn't Owner		Last Ack		Req is Owner	Last Ack
	-	-	-	-	-	-	-
M	Ill	Ack to Req Pop	Ack to Req Pop	Ill	Update L2 Ack to Req Pop	Update L2 Ack to Req, Dir is Owner, Pop	Ill
	-	-	-	-	-	-	-
E_I^{Aa}	Ack to Req Dir is Owner EtoI to Mem Pop	Ill	Ill	Ill	Update L2 Ack to Req Pop	Update L2 Ack to Req Dir is Owner, Data to Mem, Pop	Ill
	E_I^{a}	-	-	-	M_I^{da}	M_I^{a}	-
E_I^{a}	Ill	Ill	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-	-	-
S_I^{Aa}	Ill	Ack to Req Req isn't Shared, EtoI to Mem Pop	ack– Ack to Req Pop	ack– Ack to Req StoI to Mem Pop	Ill	Ill	Ill
	-	E_I^{a}	-	S_I^{a}	-	-	-
mS_I^{Aa}	Ill	Ack to Req Data to Mem Pop	ack– Ack to Req Pop	ack– Ack to Req Data to Mem Pop	ack– Ack to Req Pop	Ill	ack– Ack to Req, Data to Mem, Pop
	-	E_I^{a}	-	S_I^{a}	-	-	S_I^{a}
S_I^{a}	Ill	Ill	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-	-	-
M_I^{da}	Ill	Ack to Req Pop	Ill	Ill	Update L2 Ack to Req Pop	Update L2 Ack to Req, Data to Mem, Dir is Owner, Pop	Ill
	-	-	-	-	-	M_Ia	-
I_S^{d}	Ill	Ill	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-	-	-
I_M^{d}	Ill	Ill	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-	-	-
M_I^{a}	Ill	Ack to Req Pop	Ill	Ill	Ack to Req Pop	Ill	Ill
	-	-	-	-	-	-	-
M_S^{D}	Ill	Ack to Req Req isn't Shared, Pop	Ack to Req Req isn't Shared, Pop	Ill	Update L2 Ack to Req Pop	Update L2 Ack to Req, Req isn't Shared, Dir is Owner, Dealloc. TBE, Pop Wake Depend.	Ill

Table 2f	EtoI		StoI		MtoI		
		Req isn't Owner		Last Ack		Req is Owner	Last Ack
	-	-	-	-		S	-

Table 2g	ToM				Data_Core
		Dir is Owner/ Owner is Req	<-else	Owner isSp	
I	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-
E	Ack to Req Req isn't Shared Set MRU, Pop	Ill	Ill	Ill	Ill
	M	-	-	-	-
S	Req isn't Shared Inv (Fwd) to Non-Sp Sharers, NumAcks to Req Squash to >= first Sp Sharer, Req is Owner Set MRU, Pop	Ill	Ill	Ill	Ill
	M	-	-	-	-
M	Ill	Ack to Req Req isn't Sp Req is Owner Set MRU Pop	Fwd to Owner Req is Owner Req isn't Sp Set MRU Pop	Squash to >=min(Owner, first Sp Sharer), Inv (Fwd) to Non-Sp Sharers, NumAcks to Req Req is Owner Req isn't Sp Set MRU, Pop	Ill
	-	-	-	-	-
E_I^{Aa}	Ack to Req Req isn't Shared Pop	Ill	Ill	Ill	Ill
	M_I^{da}	-	-	-	-
E_I^{a}	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-
S_I^{Aa}	Ack to Req Req isn't Shared Reset Pending Acks Pop	Ill	Ill	Ill	Ill
	M_I^{da}	-	-	-	-
mS_I^{Aa}	Ack to Req Req isn't Shared Reset Pending Acks Pop	Ill	Ill	Ill	Ill
	M_I^{da}	-	-	-	-
S_I^{a} M_I^{da} I_S^{d} I_M^{d}	Ill	Ill	Ill	Ill	Ill
M_I^{a}	-	-	-	-	-
M_S^{D}	Stall	Ill	Ill	Ill	Update L2 Dealloc. TBE Dir is Owner Pop Wake Depend.
	-	-	-	-	S

Table 2h	InvAck		Fwd_GetM	DummyFwd	SquashReq
		Last Ack			

Table 2h	InvAck		Fwd_GetM	DummyFwd	SquashReq
		Last Ack			
I	Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
E	*Ill*	*Ill*	*Ill*	Data_NoAck to Req, Pop	Req isn't Sp Squash to > Req Pop
	-	-	-	-	-
S	*Ill*	*Ill*	*Ill*	Data_NoAck to Req, Pop	Req isn't Sp Squash to > Req Pop
	-	-	-	-	-
M	*Ill*	*Ill*	*Ill*	Data_NoAck to Req, Pop	Req isn't Sp Squash to > Req Pop
	-	-	-	-	-
E_I^{Aa}	EtoI to Mem Pop	*Ill*	*Ill*	Stall	Req isn't Sp Squash to > Req Pop
	E_I^{a}	-	-	-	-
E_I^{a}	Pop	*Ill*	*Ill*	Stall	Req isn't Sp Squash to > Req Pop
	-	-	-	-	-
S_I^{Aa}	ack– Pop	ack– StoI to Mem Pop	*Ill*	Stall	Req isn't Sp Squash to > Req Pop
	-	S_I^{a}	-	-	-
mS_I^{Aa}	ack– Pop	ack– Data to Mem Pop	*Ill*	Stall	Req isn't Sp Squash to > Req Pop
	-	S_I^{a}	-	-	-
S_I^{a} M_I^{da} I_S^{d} I_M^{d} M_I^{a}	Pop	*Ill*	*Ill*	Stall	Req isn't Sp Squash to > Req Pop
	-	-	-	-	-
M_S^{D}	*Ill*	*Ill*	Inv (Fwd) to Non-Sp Sharers, Data_Ack to Req Req isn't Sp Squash to >= first Sp Sharer, Set MRU, Pop	Stall	Req isn't Sp Squash to > Req Pop
	-	-	-	-	-

Table 2i	UpdateAsSq		Mem_Ack	Mem_Data	
		Last Ack			One Sharer Req
I	*Ill*	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
E	Dir is Owner Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
S	Req isn't Sharer Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
M	Dir is Owner if Req is Owner Pop	*Ill*	*Ill*	*Ill*	*Ill*
	-	-	-	-	-
E_I^{Aa}	EtoI to Mem Pop	*Ill*	*Ill*	*Ill*	*Ill*
	E_I^{a}	-	-	-	-
E_I^{a}	*Ill*	*Ill*	Dealloc. TBE	*Ill*	*Ill*

Table 2i	UpdateAsSq		Mem_Ack	Mem_Data	
		Last Ack			One Sharer Req
			Pop Wake Depend.		
	-	-	I	-	-
S_I^{Aa}	ack– Pop	ack– StoI to Mem Pop	Ill	Ill	Ill
	-	S_I^{a}	-	-	-
mS_I^{Aa}	ack– Pop	ack– Data to Mem Pop	Ill	Ill	Ill
	-	S_Ia	-	-	-
S_I^{a}	Ill	Ill	Dealloc. TBE Pop Wake Depend.	Ill	Ill
	-	-	I	-	-
M_I^{da}	Data to Mem Pop	Ill	Ill	Ill	Ill
	M_I^{a}	-	-	-	-
I_S^{d}	Ill	Ill	Ill	Update L2 Data_NoAck to Reqs Dealloc. TBE Pop Wake Depend.	Update L2 Data_Excl to Req Dealloc. TBE Pop Wake Depend.
	-	-	-	S	E
I_M^{d}	Ill	Ill	Ill	Update L2 Data_NoAck to Req Dealloc. TBE Pop Wake Depend.	Ill
	-	-	-	M	-
M_I^{a}	Ill	Ill	Dealloc. TBE Pop Wake Depend.	Ill	Ill
	-	-	I	-	-
M_S^{D}	Dealloc. TBE Dir is Owner Pop Wake Depend.	Ill	Ill	Ill	Ill
	S	-	-	-	-

Table 2j	{Clean} L2 Replacement (/Mem_Inv)					
		Dir is Owner	Owner isn't Sp	Owner isSp	No Non-Sp Sharers	<-else
I	Ill	Ill	Ill	Ill	Ill	Ill
	-	-	-	-	-	-
E	Ill	Ill	Inv to Owner, Alloc. TBE Dealloc. TBE, (Pop)	Squash to >=Owner, EtoI to Mem Alloc. TBE Dealloc. Block (Pop)	Ill	Ill
	-	-	E_I^{Aa}	E_I^{a}	-	-
S	Ill	Ill	Ill	Ill	Alloc. TBE {StoI}/Data to Mem, Squash to >= first Non-Sp Sharer, Dealloc. Block (Pop)	Alloc. TBE Inv to Non-Sp Sharers, Set Pending Acks Data to Mem Squash to >= first Non-Sp Sharer, Dealloc. Block (Pop)

Table 2j	{Clean} L2 Replacement (/Mem_Inv)					
		Dir is Owner	Owner isn't Sp	Owner isSp	No Non-Sp Sharers	<-else
	-	-	-	-	S_I^a	$\{S\_I^{Aa}\}/mS\_I^{Aa}$
M	Ill	Data to Mem Alloc. TBE Dealloc. Block (Pop)	Inv to Owner, Alloc. TBE Dealloc. Block, (Pop)	Squash to >= Owner, Inv to Non-Sp Sharers, Set Pending Acks Data to Mem Alloc. TBE Dealloc. Block (Pop)	Ill	Ill
	-	M_I^a	M_I^{da}	mS_I^{Aa}	-	-
E_I^{Aa} E_I^a S_I^{Aa} mS_I^{Aa} S_I^a M_I^{da} I_S^d I_M^d M_I^a	Stall	Ill	Ill	Ill	Ill	Ill
M_S^D	-	-	-	-	-	-

Appendix B

System Calls, QEMU, Kernel

This Appendix supplements Section [4.1]. In order to read or manipulate the control register *CR0*, kernel mode privileges are required. Therefore, this can be achieved by means of either writing a new kernel module, or by inserting a new system call.

The overall procedure described here will also help in setting up an environment to facilitate development of a TLS runtime that is of a more dynamic nature than what is currently being used in Section [5.1].

B.1 Using QEMU

A virtual environment achieves the best compromise between safety and speed to test out and debug changes done at the kernel level. *QEMU,* or Quick EMUlator, is a binary translation based open source machine emulator and virtualizer.

To create a loop device that will become the root filesystem, and to install *Debian*:

```
dd if=/dev/zero of=rootfs.img bs=1024 count=1048576
sudo losetup /dev/loop0 rootfs.img
sudo mkfs -t ext3 -m 1 -v /dev/loop0
mkdir mnt
sudo mount -t ext3 /dev/loop0 mnt
sudo debootstrap sid mnt http://ftp.debian.org/debian
sudo chroot mnt/ /bin/bash
#: apt-get install gcc sudo make
#: exit
sudo umount mnt
```

This *img* can be resized at any time using the *resize2fs* utility. For subsequent mounting, just the *losetup* and *mount* commands are sufficient. To enable login through a serial console, make the following changes to the *img*:

```
/etc/shadow/

      - Change root:*: to root::

/etc/inittab/

      - Uncomment the line that specifies ttyS0
```

Assuming a *bzImage* has already been compiled, *QEMU* can be invoked as follows:

```
qemu-system-x86_64 -kernel <path_to_kernel>/arch/x86/boot/bzImage \
  -hda rootfs.img -append "root=/dev/sda console=ttyS0" -nographic
login: root
#
```

B.2 Writing and Loading a Kernel Module

The following code provides a sample manipulation of *CR0* demonstrating the use of *asm*.

```
#include <linux/init.h>
#include <linux/module.h>
static int m_init(void) {
  u32 cr0, eax, ebx;
  __asm__ __volatile__ (
    "mov %%cr0, %0\n\t"  // Read CR0
  : "=r" (cr0)
  :
  :
  );
  printk(KERN_INFO "1. cr0 = 0x%8.8X\n", cr0);
  __asm__ __volatile__ (
    "mov %%cr0, %%eax\n\t"      // Move CR0 into EAX
    "or $(1 << 28), %%eax\n\t"  // Set TLS bit
    "mov %%eax, %0\n\t"
    "mov %%eax, %%ebx\n\t"
    "mov %%ebx, %1\n\t"
    "mov %%eax, %%cr0\n\t"
    "mov %%cr0, %2\n\t"
  : "=a" (eax), "=b" (ebx), "=r" (cr0)
  :
  :
  );
  printk(KERN_INFO "2. OR'd eax = 0x%8.8X\n", eax);
  printk(KERN_INFO "3. copied to ebx from eax  0x%8.8X\n", ebx);
```

```
  printk(KERN_INFO "4. copied to cr0 from eax  0x%8.8X\n", cr0);
  __asm__ __volatile__ (
    "mov %%cr0, %%eax\n\t"
    "or $(1 << 30), %%eax\n\t"  // Set Cache Disable
    "mov %%eax, %0\n\t"
    "mov %%eax, %%ebx\n\t"
    "mov %%ebx, %1\n\t"
    "mov %%eax, %%cr0\n\t"
    "mov %%cr0, %2\n\t"
  : "=a" (eax), "=b" (ebx), "=r" (cr0)
  :
  :
  );
  printk(KERN_INFO "5. Disabling cache...\n");
  printk(KERN_INFO "6. OR'd eax = 0x%8.8X\n", eax);
  printk(KERN_INFO "7. copied to ebx from eax  0x%8.8X\n", ebx);
  printk(KERN_INFO "8. copied to cr0 from eax  0x%8.8X\n", cr0);

  return 0;
}
static void m_exit(void) {
  u32 cr0, eax, ebx;
  __asm__ __volatile__ (
    "mov %%cr0, %%eax\n\t"
    "and $~(1 << 30), %%eax\n\t"  // Reset Cache Disable
    "mov %%eax, %0\n\t"
    "mov %%eax, %%ebx\n\t"
    "mov %%ebx, %1\n\t"
    "mov %%eax, %%cr0\n\t"
    "mov %%cr0, %2\n\t"
  : "=a" (eax), "=b" (ebx), "=r" (cr0)
  :
  :
  );
  printk(KERN_INFO "9. Re-enabling cache\n");
  printk(KERN_INFO "10. AND'd eax = 0x%8.8X\n", eax);
  printk(KERN_INFO "11. copied to ebx from eax  0x%8.8X\n", ebx);
  printk(KERN_INFO "12. copied to cr0 from eax  0x%8.8X\n", cr0);
  printk(KERN_INFO "Bye!!\n");
};
module_init(m_init);
module_exit(m_exit);
```

This code can be compiled into a loadable module using the following Makefile.

```
obj-m += module.o
all:
```

```
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
test:
  all
  sudo insmod ./module.ko
  sudo rmmod module
  dmesg | less  // The KERN_INFO output can be viewed this way
```

Upon installing the kernel headers into the *img* (by using a *make deb-pkg* on the kernel source and then using *dpkg -i* on the resulting *.deb* file in the *img*), a *make test* on the module gives the following (stripped) output:

```
1. cr0 = 0x8005003B
2. OR'd eax = 0x9005003B
3. copied to ebx from eax  0x9005003B
4. copied to cr0 from eax  0x8005003B
5. Disabling cache...
6. OR'd eax = 0xC005003B
7. copied to ebx from eax  0xC005003B
8. copied to cr0 from eax  0xC005003B
9. Re-enabling cache
10. AND'd eax = 0x8005003B
11. copied to ebx from eax  0x8005003B
12. copied to cr0 from eax  0x8005003B
```

As can be seen, while it is possible to modify the defined bits in *CR0*, it is not so in the case of the reserved bits.

B.3 Inserting System Calls into the Kernel

The same *asm* code for *CR0* bit manipulation can wrapped around by a system call as well, with identical results. To insert a new system call, the following files in the kernel source (as of *v*3.13.5) must be modified:

```
arch/x86/include/asm/
```

- New header file declaring the new system call as extern

```
arch/x86/include/generated/uapi/asm/unistd_32.h
```

- Assign a new ID to the new system call

`arch/x86/kernel/`

- Create a source file to define the new system call; also modify the Makefile

`arch/x86/syscalls/syscall_32.tbl`

- Append the call number ID and entry vector for the new system call

Upon re-building this source and installing the headers into the guest system, the new system call can be called from user-space code by including the *unistd_32.h* file.