Reducing Memory Buffering Overhead in Software Thread-Level Speculation

Zhen Cao Clark Verbrugge
School of Computer Science, McGill University, Montréal, Québec, Canada H3A 0E9
zhen.cao@mail.mcgill.ca, clump@cs.mcgill.ca

Abstract

Software-based, automatic parallelization through Thread-Level Speculation (TLS) has significant practical potential, but also high overhead costs. Traditional "lazy" buffering mechanisms enable strong isolation of speculative threads, but imply large memory overheads, while more recent "eager" mechanisms improve scalability, but are more sensitive to data dependencies and have higher rollback costs. We here describe an integrated system that incorporates the best of both designs, automatically selecting the best buffering mechanism. Our approach builds on well-optimized designs for both techniques, and we describe specific optimizations that improve both lazy and eager buffer management as well. We implement our design within MUTLS, a software-TLS system based on the LLVM compiler framework. Results show that we can get 75% geometric mean performance of OpenMP versions on 9 memory intensive benchmarks. Application of these optimizations is thus a useful part of the optimization stack needed for effective and practical software TLS.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–Parallel programming; D.3.4 [*Programming Languages*]: Processors–Run-time environments

Keywords Software thread-level speculation, automatic parallelization, memory buffering, optimization

1. Introduction

Traditionally a hardware-oriented technique, thread-level speculation (TLS) has received extensive research attention [4, 6, 16, 20]. However, hardware TLS tends to be constrained to small granularity parallelism due to limited hardware buffering resources. A tight analysis on the SPEC CPU2006 benchmarks showed that the speedup potential uniquely achievable by TLS at the innermost loop level is the order of 1% [8], suggesting TLS needs be applied at a larger granularity to be more effective [5].

Software approaches to TLS have been explored as well, primarily for their huge advantage in applying to existing, commodity hardware [12, 15, 23]. Software TLS also has the advantage of much greater and more flexible resource limits, especially in terms of memory. This potentially allows for larger granularity in the parallelism. The actual use of such resources, however, has been blamed for limiting scalability [13]. The large buffering costs inherent in the design of software isolation and validation mechanisms result in significant memory traffic, impacting cache performance and resulting in long validation/commit (V/C) times.

Two approaches to software TLS buffering have been proposed: lazy version management and eager version management (in-place update) [13, 23]. Most software TLS systems, such as SableSpMT [15], SpLSC [12], Lector [23] and MUTLS [3], adopt the lazy version management approach, which buffers data accessed by speculative threads and employs a serial commit phase to validate/commit the speculative buffer to main memory. If there are a significant number of threads running, the serial commit phase may become a scalability bottleneck for memory intensive applications due to it delaying the critical path. Recent software TLS systems such as SpLIP [13] and MiniTLS [23] apply eager version management to address the problem, allowing speculative threads to directly access main memory and thereby eliminating the serial commit phase. In this approach, history versions of accessed data are maintained in shadow buffers and are used to recover the main memory state when rolling back offending threads. However, despite the advantage of higher scalability, eager version management has the disadvantage of causing rollbacks for all RAW, WAR and WAW dependencies, as well as incurring expensive rollback overhead if dependency occurs.

Since lazy and eager version management have complementary weaknesses and strengths, it would be more effective to integrate them into one system to get the strengths of both [13]. Our paper proposes the first such software TLS buffering solution which can automatically determine which version of management buffering to apply to which variables. This approach is integrated with and complemented by a number of other optimizations that reduce buffer size and improve buffer management. More specifically, we have the following contributions.

To improve lazy version management, we propose a per-thread page-table memory buffering scheme that enables direct parallelization of the V/C operations themselves. Our parallelized V/C takes advantage of extra processors still available once scalability has been saturated in order to reduce the overhead of one of the more important overhead concerns in lazy buffering schemes. We also use vector processing to accelerate both address-space checking and memory-buffering V/C through common SIMD instructions, demonstrating application of both coarse and fine-grain parallelism as a means of helping the TLS system itself, and indicate an interesting optimization point exists in balancing the parallel resources applied to the base program with the resources used to implement that parallelism.

For eager version management, we describe a design for shared address-owner memory buffering where the space overhead is bounded by a constant factor of the program data size. The size

CC'16, March 17–18, 2016, Barcelona, Spain
ACM. 978-1-4503-4241-4/16/03...
http://dx.doi.org/10.1145/2892208.2892227

of shadow buffers is a significant concern in eager approaches, and can limit the ability to exploit parallelism at larger granularity. Our design allows buffers to be allocated sufficiently large to enable speculation of any granularity without causing buffering overflow.

Significant reductions to data management costs are also possible if we know data is not changed at runtime, and so does not require temporary buffer space or need to participate in validation. Pure readonly data is uncommon and difficult to find in programs, but in a TLS context opportunities are improved by the fact that we only require variables be readonly during actual periods of overlapping speculative executions, and this has been the basis of previous approaches that accurately find readonly variables with the help of profiler support [5] or through manual programmer specifications [12]. We describe a design that automates the process, using page markers to identify and optimize readonly and independent memory pages on-the-fly, giving us a less precise but low overhead and fully dynamic means of finding readonly data.

Finally, we propose a buffering integration mechanism that can automatically select the appropriate buffering technique for each variable. It can quickly identify variables as independent (without RAW, WAR, WAW dependencies, *i.e.* readonly or access disjoint memory) or not, and apply the optimized address-owner buffering for independent variables and the page-table buffering for dependent ones. In this way, we can benefit from the higher scalability of eager version management for independent variables, while still enabling TLS in the presence of dependent ones in a speculative region. This design includes adaptive buffering selection heuristics to dynamically choose the appropriate buffering based on the program execution characteristics, as well as a thread stopping optimization to improve thread coverage.

We implement and evaluate our techniques within MUTLS, a full-featured, LLVM-based software TLS system [1, 3]. With only fork point annotation of the MUTLS system, we observe these optimizations result in significantly higher speedups, achieving 75% geometric mean performance of the OpenMP version on 9 memory-intensive benchmarks.

2. Related Work

Memory buffering has been the concern of many researches on software-TLS and other speculative parallelization approaches.

Oancea et al. [13] proposed SpLIP, a lightweight, in-place update (eager version management) software-TLS approach to achieve higher scalability. We make use of this general technique as well, exploring it in conjunction with an optimized lazy version management scheme that also improves scalability, rather than as a direct replacement.

Rundberg and Stenstrom [19] put forward a software-TLS system that detected dependency and forwarded data on-the-fly during speculative loads/stores, enabling *parallel commit* after speculative execution. The parallel commit design explores parallelism at a coarser granularity than parallelized V/C, but the approach may have impractically large memory space overhead in the presence of aliasing [13]. Yiapanis et al. [23] presented a compact version management data structure and applied it to propose an eager and a lazy version management software-TLS systems, MiniTLS and Lector, achieving average speedups of 7x and 8.2x respectively on a 32-core machine. The rollback procedure of MiniTLS was parallelized, while the serial commit procedure of Lector was not.

Several speculative parallelization systems proposed efficient memory buffering load/store implementations. BOP [5] uses processes instead of threads for data protection. It enables *strong isolation* where memory overhead is proportional to the data size accessed instead of the number of data accesses, which benefits applications with high temporal locality. Tian et al. [21] proposed the Copy-or-Discard (CorD) speculative execution model that finds multi-versioned variables using a mapping table, and optimized it for dynamic pointer-intensive data structures [22]. LSA-STM [18] improves lazy version management for object-based software transactional memory (STM) with eager ideas, which uses validity ranges to avoid revalidating previously read objects for each new read.

Parallelism of the runtime system has also been explored by STM approaches. STMlite [11] removes lock overhead of transactional execution by centralizing transaction bookkeeping in a single core so that it runs in parallel with work threads. Raman et al. [17] proposed a Software Multi-threaded Transaction (SMTX) system that exposes data parallelism by dividing loop iterations to different pipeline stages while still committing them together. SMTX uses a separate centralized commit process to reduce inter-core communication latency on the critical path.

BOP [5] and SpLSC [12] proposed optimizations for readonly shared variables, but required profiling tools support or programmer specification to find likely-readonly variables. We propose the readonly-page optimization to automatically find and optimize readonly variables on-the-fly during speculative execution. Oancea et al. [13] point out that integrating different memory buffering approaches should be more beneficial than single buffering implementations. We provide the first buffering integration solution, which seamlessly integrates different buffering or pseudo-buffering implementations into one buffering framework and automatically utilizes the most appropriate one for each variable.

DynTM [9] and SEL-TM [24] are hardware transactional memory (HTM) systems with hybrid conflict management policies that permit eager and lazy version management to cooperate during transactional execution. ASTM [10] and adaptSTM [14] use heuristics to select eager or lazy version management for software transaction memory (STM) threads. We propose the first software-TLS solutions to allow each speculative thread to utilize both eager and lazy buffering.

Garzaran et al. [7] analyzed complexity-benefit tradeoffs of different hardware-TLS memory buffering approaches. They discussed strengths and weaknesses as well as the required hardware support for each approach, and compared the mechanisms using 7 benchmarks with different memory access patterns. Our work improves software-TLS buffering approaches and integrates them into a unified framework with the aim to maximize their combined strengths.

3. Background

Thread-level speculation (TLS) is an optimistic approach to automatic parallelization, allowing the compiler to exploit parallelism from programs without the need to prove absence of actual, or even potential dependencies. A speculative thread is launched at a *fork point*, executing from a *join point* well ahead in terms of sequential execution. Memory reads and writes of the speculative thread are buffered to maintain correctness. When the non-speculative thread (representing sequentially earliest execution) reaches the join point, the speculative thread validates its read buffer; if no dependency occurred, it commits its write buffer to memory and merges its execution state to the non-speculative thread; otherwise, it is rolled back and re-executed by the non-speculative thread.

There are different approaches to TLS memory buffering. Garzaran et al. [7] proposed a 2-dimensional taxonomy of hardware-TLS buffering approaches. One dimension is separation of speculative task state within a CPU and the other is merging of speculative task state to the main memory. The former deals with the issue of reducing CPU idle time to improve speculative work time coverage, which is beyond the scope of the paper. The latter has three categories: Eager AMM (Architectural Main Memory), Lazy AMM and FMM (Future Main Memory). AMM buffers speculative state in the CPU and commits it to the main memory after a speculative thread completes execution, while FMM directly accesses memory and buffers the memory data, which is used to restore the main memory state if the speculative thread rolls back. Eager AMM commits all buffered data at commit time while Lazy AMM only commits a cache line when another speculative thread uses it again. Lazy and eager version management software-TLS buffering implements Eager AMM and FMM, respectively, while Lazy AMM has not been implemented by software-TLS.

Two mechanisms to implement software-TLS lazy buffering have been proposed, based on there being a non-speculative thread [3, 15] or only speculative ones [12, 19, 23]. Both have their own advantages: the former does not buffer the non-speculative thread and thus can guarantee worst-case run time excluding threading overhead, while the latter buffers memory accesses of all threads which enables better optimizations for the speculative threads. In our work we use lazy version management with the non-speculative thread, the architecture of which is illustrated in Figure 1.

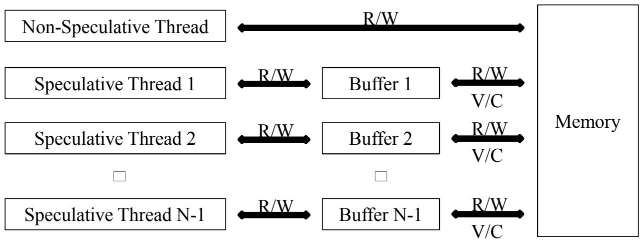


Figure 1. Lazy Version Management Buffering

Existing software-TLS eager buffering mechanisms such as SpLIP [13] and MiniTLS [23] maintain a shadow buffer for each speculative thread, and add a new version to the buffer each time a variable is written, as per an FMM implementation. The architecture is shown in Figure 2.

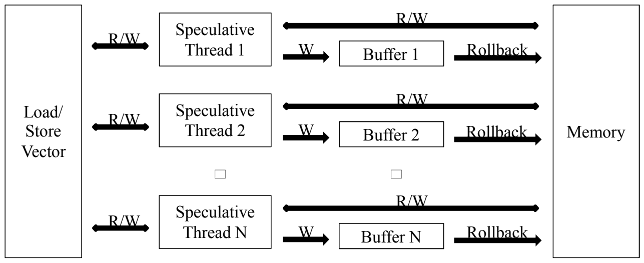


Figure 2. Existing Eager Version Management Buffering

3.1 MUTLS

MUTLS (Mixed-model Universal software-TLS) [3] is a language and architecture independent software-TLS system based on the well-defined LLVM intermediate representation (IR). LLVM is a popular compiler infrastructure with many powerful analysis and transformation passes for program optimization. Since the MUTLS transformation pass is a purely LLVM-IR based pass (from well-formed LLVM IR to well-formed LLVM IR), it is fully integrated into the LLVM compiler framework, which can take advantage of full optimizations as well as all source languages and target architectures enabled by the LLVM framework. With a mixed forking model, MUTLS is able to exploit more parallelism from tree-form recursion applications.

MUTLS has two types of threads: a non-speculative thread and speculative threads. The non-speculative thread represents logically earliest execution that never rolls back and is not buffered. Memory accesses of speculative threads are buffered and causes the offending threads to rollback if validation detects RAW dependencies at thread join time. Since speculative threads are usually slower than the non-speculative thread due to buffering cost, checkpoints are inserted in loops and before nested function calls so that a speculative thread can be joined whenever needed, which guarantees that the software-TLS system is efficient even for memory-intensive applications. This feature also leads MUTLS to be an arbitrary-point speculation system [3, 5, 16] that has more parallelism potential than loop-level speculation and method-level speculation.

MUTLS is comprised of a front-end, an LLVM transformation pass and a TLS runtime library. The front-end annotates *fork/join/-barrier points* with LLVM built-in functions to specify where to fork/join/barrier speculative threads, as specified manually by programmers, or automatically by the compiler, profilers, or other tools. A speculative thread is created at a fork point, starts execution from the corresponding join point, is joined (merged) when the non-speculative thread reaches the join point, and barriered at a corresponding barrier point if the speculative thread reaches it. The runtime library defines API calls for certain behaviours such as forking/joining threads and buffering loads/stores. The transformation pass transforms the incoming IR based on the annotated fork/join/barrier points and delegates specific speculation behaviours to the TLS runtime library.

4. Memory Buffering

In this section, we describe our optimized designs for lazy and eager buffering approaches and their adaptive integration. First, we present the page-table memory buffering for lazy buffering, which allows us to exploit both coarse and fine grain parallelism in the validation/commit phase. Next, we describe the shared address-owner buffering that enables higher scalability and reduced buffer overflow in an eager design. Readonly data detection, adaptive buffering and thread stopping optimization are related in their integration, and so are described together at the end of this section.

Note that we adopt page-based designs to both the buffering integration mechanism and the page-table thread memory buffering implementation. These pages, however, are independent of each other and have different characteristics: the former need only improve performance in the most common cases and thus is not required to be accurate, while the latter is expected to support a wide range of applications as long as they do not exhibit true RAW dependencies, and hence accurate, one-to-one mapping of each byte from the main memory to the buffering is important. The two page-based designs also serve different purposes. The former is to reduce the TLS system overhead by maintaining optimization meta-data at a coarser granularity with page-based data structures. The latter exposes coarse and fine grain parallelism to reduce validation/commit time for the thread memory buffering.

4.1 Lazy Per-Thread Page-Table Buffering

The basic MUTLS buffering approach does not buffer the non-speculative thread, following a lazy version management approach discussed in section 3. Page-table thread memory buffering implements this same behaviour, but organizing data to facilitate parallelization in the final V/C stage of each speculative thread. Note that these page-table buffers are thread-specific, and thus synchronization between speculative threads is not needed.

The page-table thread memory buffering maintains a page table, a read-set and a write-set. A memory store directly inserts the address-value pair to the write-set, while a memory load returns the buffered data from its write-set or read-set if found, and otherwise inserts the address and the read data to the read-set and returns the data. During validation, if the buffered data in the read-set is not equal to the main memory version, then RAW dependencies are detected and the thread is rolled back; otherwise, validation

```
class ThreadBuffer{
    int rank;
    char* table[PAGE_NUM];
    array<size_t, PAGE_NUM> pages;
    char markR[SIZE], bufR[SIZE];
    char markW[SIZE], bufW[SIZE];
public:
    T load(T* addr);
    void store(T* addr, T data);
    void commit(){
        for(each p in pages)
            commit_page(p);
    }
    bool validation(){
        for(each p in pages){
            if(!validate_page(p))
                return false;
        }
        return true;
    }
};
```

```
size_t find_page(size_t addr){
    size_t n = PAGE_NUM / K;
    size_t index = (addr / PAGE) % n;
    for(; index < PAGE_NUM; index += n){
        if(table[index] == addr) return index;
        if(table[index] == NULL){
            table[index] = addr;
            pages.push_back(index);
            return index;
        }
    }
    rollback(OVERFLOW, rank);
}
```

```
template<typename T>
void store(T* addr, T data){
    size_t p = find_page((size_t)addr);
    size_t ofs = (p * PAGE) + (addr % PAGE);
    *(T*)(markW + ofs) = (T)all_one;
    *(T*)(bufW + ofs) = data;
}
```

```
template<typename T>
T load(T* addr){
    size_t p = find_page((size_t)addr);
    size_t ofs = (p * PAGE) + (addr % PAGE);
    if(*(T*)(markW + ofs) == (T)all_one)
        return *(T*)(bufW + ofs);
    if(*(T*)(markW + ofs) != 0)
        rollback(PART_ACCESS, rank);
    if(*(T*)(markR + ofs) == (T)all_one)
        return *(T*)(bufR + ofs);
    if(*(T*)(markR + ofs) != 0)
        rollback(PART_ACCESS, rank);
    *(T*)(markR + ofs) = (T)all_one;
    *(T*)(bufR + ofs) = *addr;
    return *addr;
}
```

rank: the rank of the thread (1 to N_THREAD-1)
K: associativity of the hash mapping (4)
SIZE: the size of the thread buffer (512MB)
PAGE: the size of each page (4KB)
PAGE_NUM = SIZE / PAGE

Figure 3. Lazy Per-Thread Page-Table Memory Buffering

succeeds and the thread commits all buffered data in the write-set to main memory.

The page-table thread memory buffering implementation is detailed in Figure 3. The buffering maintains a page table (`table`, `pages`), a read-set (`markR`, `bufR`) and a write-set (`markW`, `bufW`). The thread memory buffering is organized as pages of *PAGE* bytes. The offset *ofs* of each byte of the read-set and write-set can be calculated as $ofs = i * PAGE + o$ given page index i and the offset o within the page. To support the case that multiple addresses are hashed to the same page, a K-way associative hash mapping is implemented, which considers the buffer to comprise K consecutive blocks and each address can be hashed to any of the K blocks. Each read-/write-set page is comprised of a *data page* to buffer memory data and a *mark page* to indicate which bytes of each data page are accessed. The `load` method first finds the page index and then computes the buffer offset of the address. If the address is fully buffered in the write-set or read-set, the data is returned. If part is buffered, it means the program uses aliasing of different data types, which we consider rare and in which case we simply rollback the thread. If the data is not buffered, it is inserted into the read-set and returned.

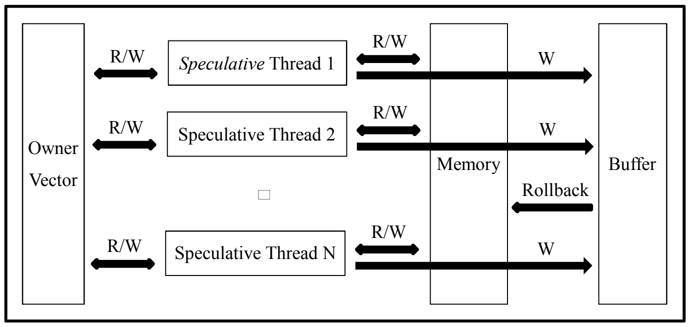
The point of this design is that validation/commit within a page can be vectorized and on different pages can be parallelized, which helps to achieve scalable speedups for memory intensive applications. To enable SIMD acceleration and/or parallelized V/C, it should be guaranteed that validation/commit within and/or across pages are independent and not subject to sequential ordering, which is the case for the page-table memory buffering. Other software-TLS approaches such as SpLSC [12] and Lector [23] do not enable these optimizations since in their designs multiple writes to the same address by a speculative thread are all stored in the write-set, and therefore must be committed serially to guarantee the last write is the last to commit. Also, the addresses in the write-set are not necessarily adjacent.

In addition to exposing different granularities of parallelism, the page-table memory buffering has other advantages over previous software TLS buffering implementations. First, the memory buffer can accurately track dependencies with mixed load/store data types. Different threads accessing adjacent memory locations will not cause thread rollbacks, even for those accessing different integers or characters within the same 8-byte boundary on a 64-bit machine. Second, the K-way associative hash mapping can effectively reduce rollbacks caused by hashing conflicts for programs with many variables such as memory allocation/deallocation intensive applications. However, we also note that these advantages come at costs. To support accurate dependency tracking, the average buffering space overhead for each word is $2 + \epsilon$ words for read-only or write-only variables and $4 + \epsilon$ for read-write variables, where ϵ is the paging overhead, compared to 2 (a buffered word and an address) for SpLSC and Lector. The K-way hash mapping also incurs performance overhead. Nevertheless, we find the benefits outweigh the overhead, as the page-table buffering is utilized in the fallback path expected to support general cases.

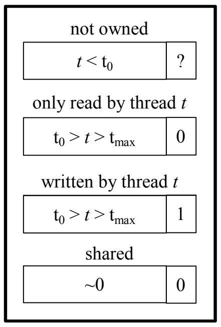
4.2 Eager Shared Address-Owner Buffering

The address-owner buffering is an eager version management memory buffering illustrated in Figure 4(a). Notably, our design uses a single shadow buffer that is shared by all threads, and at most one buffering copy is maintained for each main memory address. Therefore given the amount of accessed data (D) and the number of memory stores (W) of the original sequential program, the space complexity of the shared address-owner buffering is $O(D)$, as opposed to $O(D + W)$ for typical eager buffering designs shown in Figure 2. This ensures that the space overhead of the buffering is bounded by a constant factor of the amount of program data, and further enables optimizations that can generally assume the buffering is sufficiently large that it has a one-to-one mapping of the main memory for any parallelism granularity, since allocating more memory by a constant factor is usually not a severe problem. As a result, K-way hash mapping, and thus the page table, is not needed. However, we need to include the non-speculative thread (Thread 1, with "Speculative" italicized) in the buffering design, in order to be able to detect interference between speculative threads and the non-speculative thread.

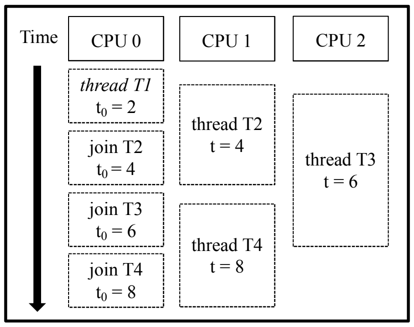
The granularity of the buffering is WORD; different threads accessing the same WORD with at least one writing are considered to have dependencies. WORD can be set as the native word of the machine, or smaller to achieve higher precision of dependency tracking. By our design, using smaller WORD such as 8, 16 or 32 bit is still as efficient as using larger WORD such as 64 bit for 64-bit memory accesses, though smaller WORD overflows earlier as more threads are speculated and thus requires more frequent buffer flushes. However, for relatively large WORD such as 32-bit, this rarely occurs. For each WORD, the buffering maintains an *owner* for dependency tracking and a shadow memory copy for rolling back the WORD if dependency occurs. The encoding of the owner WORD is illustrated in Figure 4(b): the last bit of the owner WORD



(a) Architecture



(b) Owner WORD



(c) Timeline of thread owner numbers

```
1   class AddressOwnerBuffer{
2       char owners[SIZE], buf[SIZE];
3       WORD thread_owners[N_THREAD], counter;
4   public:
5       bool register_load(WORD* addr, int rank);
6       bool register_store(WORD* addr, int rank);
7       void register_start_thread(int rank){
8           counter += 2; thread_owners[rank] = counter;
9       }
10      void register_join_thread(int rank){
11          thread_owners[0] = thread_owners[rank];
12      }
13      void rollback_page(WORD* addr, size_t size){
14          WORD* p = (WORD*)(owners + (addr & SIZE)),
15          *q = (WORD*)(buf + (addr & SIZE));
16          for(size_t i = 0; i < size / sizeof(WORD); i++){
17              if(p[i] > thread_owners[0] && (p[i] & 1) == 1)
18                  addr[i] = q[i];
19              p[i] = 0;
20          }
21      }
22  };
```

```
23  bool register_load(WORD* addr, int rank){
24      WORD* p = (WORD*)(owners + (addr & SIZE)), owner = *p, t = thread_owners[rank];
25      const intptr_t SHARED = ~1LL;
26      if(owner == SHARED || (owner & SHARED) == t) return true;
27      if((owner & 1) == 0 || owner < thread_owners[0]){
28          T new_owner = (owner < thread_owners[0] ? t : SHARED);
29          T o = __sync_val_compare_and_swap(p, owner, new_owner);
30          if(o == owner) return true;
31          if((o & 1) == 0 && (__sync_val_compare_and_swap(p, o, SHARED) & 1) == 0) return true;
32      }
33      return false;
34  }
35  bool register_store(WORD* addr, int rank){
36      WORD* p = (WORD*)(owners + (addr & SIZE)), owner = *p, t = thread_owners[rank];
37      if(owner == (t | 1)) return true;
38      if((owner < thread_owners[0] || owner == t) &&
39              __sync_bool_compare_and_swap(p, owner, t | 1)){
40          *(WORD*)(buf + ofs) = *addr; return true;
41      }
42      return false;
43  }
```

SIZE: the size of the shared address owner buffer (4GB)

(d) Implementation

Figure 4. Eager Shared Address-Owner Memory Buffering

is 1 if the buffering WORD has been written by a thread and thus is exclusively owned by the thread and 0 otherwise. Higher bits denote the thread accessing the WORD (the owner thread number of the WORD), or an all-one value if it is read by more than one thread (shared), in which case the last bit is 0. The non-speculative thread has the lowest thread owner number t_0. If the owner number t of the WORD is smaller than t_0, the WORD is not owned by any actively running thread (accessed by committed threads or not accessed by any thread); if t is the owner number of a running thread, the WORD is owned by the thread; if t is ~ 0, the WORD is shared (can be read by all running threads).

To identify owner threads for buffering WORDs, globally unique thread owner numbers are needed. Since MUTLS is an arbitrary point TLS system, in which fork/join points can be inserted anywhere in a function, it is not possible to use loop iteration numbers to identify threads. The non-speculative thread also joins the speculative threads [3], and thus needs to update its thread owner number to be able to access the WORDs owned by committed threads. To resolve these issues, we maintain an owner number for each thread through a global counter: each time a thread is speculated, we increment the counter and assign it as the owner number of the speculative thread. We use the in-order forking model of the MUTLS system (only the most speculative thread can fork a new thread) and thus the non-speculative thread has the lowest owner number of the actively running threads. A running instance is demonstrated in Figure 4(c). The non-speculative thread T1 runs on CPU 0. The speculative threads T2, T3 and T4 are in-order speculated running on CPU 1, 2 and 1, respectively. The non-speculative thread T1 then joins the threads in the same order. It can be noted that the thread on CPU0 always has the lowest owner number among the running threads (horizontally) anywhere in the (vertical) time-line.

The first time an address is written, the memory data is copied to the shadow buffer, and is restored to the main memory if the thread rolls back. It should also be guaranteed that the data *only* be copied the first time it is accessed, since we only maintain one version of each buffering WORD and should ensure the buffered data be the original memory version not written by any speculative thread. This enables the shared address owner buffering to use a single global shadow buffer with larger hash space, as opposed to the multiple buffer architecture shown in Figure 2.

The shared address-owner memory buffering implementation is presented in Figure 4(d). Thread owner numbers of speculative threads are assigned by `register_new_thread` and the non-speculative thread owner number `thread_owners[0]` is updated by `register_join_thread`. The owner and shadow memory copy WORDs are stored in `owners` and `buf`, respectively. The `register_load` method returns true to indicate the load is valid if the buffering WORD of the address `addr` is shared or is already owned by the thread (Line 26). Otherwise it registers the owner if the buffering WORD is not exclusively owned by another thread (Line 27–30); if it fails, then another thread is simultaneously registering the owner, in which case it tries to register the owner as shared if the buffering WORD is not registered as exclusively owned by another thread (Line 31). If it cannot register a valid owner, false is returned (Line 33) and rollback is initiated. The `register_store` method returns true if the buffering WORD is already exclusively owned by the thread (Line 37). Otherwise, it tries to register its owner (Line 38–39) and copy the memory value to the shadow buffer (Line 40). No other threads can register ownership of the buffering WORD again after the thread registers itself as an exclusive owner, thus guaranteeing the copy-only-once requirement of the buffering.

If a thread detects dependencies in the shared buffering, all speculative threads are rolled back, and then buffering data is restored to the main memory, as illustrated by the `rollback_page` method. If a speculative thread is not rolled back due to the shared buffering dependency, e.g. attempting to call I/O functions, other speculative threads are not required to rollback since they cannot access variables written by the rolling back thread, though speculative threads representing sequentially later execution may need cascade rollback. When such a speculative thread is rolling back, it instead calls the `rollback_page_sp` method, whose implementation is similar, except that it checks if the WORD owner number is equal to the speculative thread owner number instead.

4.3 Readonly-Page Optimization & Buffering Integration

Speculative regions often contain readonly variables, and this can include larger data structures such as matrices, trees, and graphs that consume significant buffer space and thus validation time. Aggressively pruning out readonly data is thus worthwhile, and can be efficiently done at the page level. Our readonly-page optimization maps the address of each read to its address and optimizes the read by not buffering it if the page address is marked readonly. If a thread then attempts to write an address whose page is marked readonly, all speculative threads are rolled back.

In order for the optimization to not cause many rollbacks, if one page address of a variable is written by a thread, we mark *all* pages of the written variable as not-readonly, which can usually find all readonly and not-readonly pages within short amount of time after entering a loop speculative region. One issue to be considered is pointer-based heap data structures such as trees and graphs. If we consider different nodes to be different variables, the optimization would be ineffective: if a tree is not readonly, writing a node could only mark the single node not-readonly and thus each node may cause one rollback. We solve this problem by treating each heap allocation call program instruction as a single heap variable since the nodes of a data structure are usually allocated by the same instruction.

Our readonly-page optimization actually builds on a more general buffering mechanism that identifies pages as either readonly (2), independent (1), or dependent (0), using this distinction to help drive the choice of eager or lazy buffering. Before entering a speculative region, all pages are optimistically set to the default type (2, or readonly). If an address causes rollback and its page type is not at 0 (dependent), all pages of the variable that contains the address are reduced to a lower type. Figure 5 includes an example: the variable A spans two pages 101 and 102, which are initialized to 2 before entering the speculative region. After entering the speculative region, 3 speculative threads T1, T2 and T3 read A, and as long as none write A the TLS runtime system assumes the variable is readonly and does not buffer it. If A is written the threads are rolled back and restarted with A set to be independent and buffered using eager version management buffering. This continues as long as T1, T2 and T3 only read each given word or read/write different words of A. When different threads write the same word of A, threads again rollback and restart with A set as dependent using the lazy version management buffering.

To prevent the eager buffering from slowing down the non-speculative thread during non-parallelizable region, we use the sequential region optimization that the non-speculative thread checks at each memory access if it is in a speculative region, and if not then skips the buffering integration mechanism and directly accesses main memory.

The memory buffering integration mechanism, which is also the top-level implementation of the memory buffering, is present in Figure 5(b). To efficiently find the variable of an address and all pages of a variable/memory allocation instruction, we maintain the address and size of each variable and the allocated variables of each memory allocation instruction. For each page, we also register all variables accessing the page. We register variable pages in the buffering (Line 14–20) and reset the all pages before entering speculative regions (Line 22). When a speculative thread accessing an address of a non-0 type page causes rollback, it sets the `rbk_addr` to the address and the `rbk_page_type` to the appropriate lower type and then rolls back itself (Line 38). When the non-speculative thread accesses an invalid page, it aborts all speculative threads, rolls back the eager version management address-owner buffering and resets the page types of the variable that contains the address (Line 41–46). The non-speculative thread also frequently calls `check_valid` and initiates thread rollbacks if it finds that a speculative thread has accessed invalid pages (Line 32–34).

One may argue that the readonly-page optimization is not a significant contribution since accelerating readonly variables dynamically was proposed in [12] and applied in [13]. However, we note that their approach has several drawbacks. First, each memory load/store must be explicitly annotated with specific memory ranges, which is tedious when done manually for large programs and is difficult automatically without ahead-of-time profiling, while the readonly-page optimization identifies readonly variables on-the-fly automatically. Second, since memory accesses must be associated with memory ranges, their approach is generally ineffective for pointer-based data structures. Third, the annotated memory ranges must be disjoint during whole program execution, while the readonly-page optimization can adapt to different readonly variables each time a speculative region is entered.

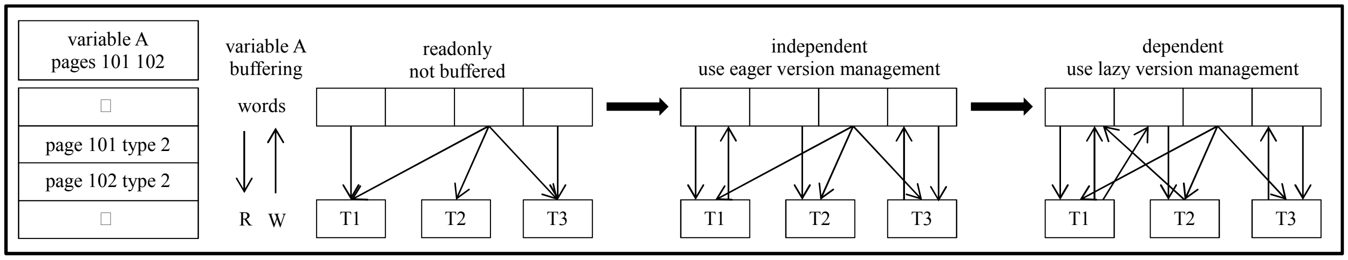
4.3.1 Adaptive Buffering Selection Heuristics

Since eager version management buffering has higher scalability, the buffering integration mechanism defaults to selecting the eager shared address-owner buffering whenever possible. However, when there are few processor cores, or the validation/commit time is small, the way eager buffering delays the non-speculative thread becomes the dominant overhead, and it can be more efficient to use the lazy page-table buffering. We thus propose adaptive buffering selection heuristics to address this problem.

The adaptive buffering selection heuristics is similar to adaptive fork heuristics [2] in that it dynamically profiles the parallel program execution and adapts to the appropriate buffering on-the-fly. At the beginning of program execution, the lazy page-table buffering is the default selection since it does not cause unnecessarily rollbacks. The runtime system records the number of memory accesses m, work time T_{work} (the time from being speculated to the start of validation/commit), and validation/commit time T_{vc} of each speculative thread. When a speculative thread commits, the expected runtime of lazy and eager buffering are estimated as follows. Assume each speculative memory access has overhead C cycles and the eager buffering delays each thread by a constant factor of K: if there are $N - 1$ speculative threads, the work time speedup of the N threads for the lazy buffering is $S = 1 + (N - 1) * (T_{work} - C * m)/T_{work}$. For an assumed workload of L loop iterations, the estimated runtime of the lazy and eager buffering is then $t_{lazy} = L * T_{work}/S + L * T_{vc}$ and $t_{eager} = L * T_{work} * K/N$, respectively. If $t_{lazy}/t_{eager} = N/K * (1/S + T_{vc}/T_{work}) > 1$, then the eager buffering is considered more efficient and should be enabled. In our experiments, we use parameters $C = 20$ and $K = 8$.

4.3.2 Thread Stopping Optimization

In the original MUTLS threading design as was summarized in section 3.1, each thread is bound to a virtual CPU that always corresponds to an operating system (OS) thread. The number of virtual



(a) Buffering example. Threads T1, T2 and T3 read/write variable A, which has pages 101 and 102.

```
class MemoyBuffering{
    int page_types[PAGE_NUM], rbk_page_type;
    char* page_addresses[PAGE_NUM], *rbk_addr;
    array<size_t, PAGE_NUM> pages;
    ThreadBuffer threads_buffer[N_THREAD];
    AddressOwnerBuffer owner_buffer;
    size_t get_page(char* addr){ return (addr & SIZE) / PAGE; }
    int get_page_type(char* addr){
        size_t p = get_page(addr), paddr = addr & ~(PAGE □1);
        return (page_addresses[p] == paddr) ? page_types[p] : 0;
    }
public:
    void register_variable_node(char* addr, size_t size){
        size_t p = get_page(addr); pe = get_page(addr + size)
        for(; p <= pe; p++, addr += PAGE){
            if(page_addresses[p] == NULL){
                page_addresses[p] = addr & ~(PAGE □1);
                pages.push_back(p);
            }
        }
    }
    void reset(){ for(each p in pages) page_types[p] = 2; }
    T load(T* addr, int rank);
    void store(T* addr, T data, int rank);
    void register_start_thread(int rank){
        owner_buffer.register_start_thread(rank);
    }
    void register_join_thread(int rank){
        owner_buffer.register_join_thread(rank);
    }
    void check_valid(){
        if(rbk_addr == NULL) return;
        rollback_self_nonsp(addr, rbk_page_type);
        rbk_addr = NULL;
    }
};
void rollback_self_sp(char* addr, int type, int rank){
    rbk_addr = addr; rbk_page_type = type; rollback(INVALID_PAGE, rank);
}
void rollback_self_nonsp(char* addr, int type){
    rollback_all_speculative_threads();
    var = find_variable(addr);
    for(each p in var.get_pages()){
        page_types[p] = type;
        owner_buffer.rollback_page(page_addresses[p], PAGE);
    }
}
template<bool sp, typename T>
T load(T* addr, int rank){
    if(is_self_stack_address(addr, rank)) return *addr;
    int t = get_page_type(addr);
    if(t == 2) return *addr;
    if(t == 1){
        if(owner_buffer.register_load(addr, rank)) return *addr;
        else sp ? rollback_self_sp(addr, 0, rank) : rollback_self_nonsp(addr, 0);
    }
    return sp ? threads_buffer[rank].load(addr) : *addr;
}
template<bool sp, typename T>
void store(T* addr, T data, int rank){
    if(is_self_stack_address(addr, rank)){ *addr = data; return; }
    size_t t = get_page_type(addr);
    if(t == 2) sp ? rollback_self_sp(addr, 1, rank) : rollback_self_nonsp(addr, 1);
    else if(t == 1){
        if(owner_buffer.register_store(addr, rank)) *addr = data;
        else sp ? rollback_self_sp(addr, 0, rank) : rollback_self_nonsp(addr, 0);
    }
    else if(sp) threads_buffer[rank].store(addr, data); else *addr = data;
}
```

SIZE: the size of the shared address owner buffer (4GB)
PAGE: the size of each page (4KB)
PAGE_NUM = SIZE / PAGE

(b) Implementation

Figure 5. Buffering example. Threads T1, T2 and T3 read/write variable A, which has pages 101 and 102.

CPUs is usually set to be no more than the number of physical CPUs of the running machine to avoid performance degradation caused by threads representing sequentially later execution competing with the CPU time of sequential earlier threads. This design however, may unnecessarily limit thread work coverage of the software-TLS system, as demonstrated by the example in Figure 6.

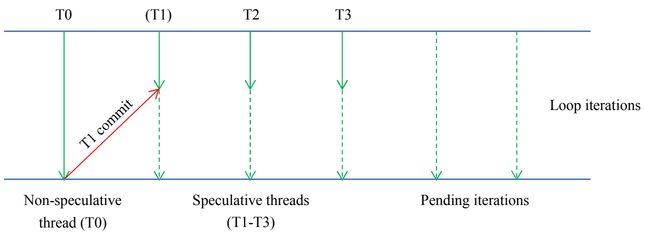


Figure 6. Thread Coverage Problem of MUTLS Threading Design. After T1 commits, only 3 threads are running.

A loop with N iterations is speculatively parallelized with P virtual CPUs ($N > P$). In the figure, P is 4 and the non-speculative thread T0 in-order speculated threads T1, T2 and T3. When T0 completes its iteration, it joins T1 and T1 commits, after which there are only 3 threads running. As the speculative threads need buffering and thus are slower than the non-speculative thread, more time is required before T3 reaches the end of the iteration to speculate a new thread, resulting in less than optimal thread coverage.

We propose the thread task optimization to solve the problem. Instead of binding each thread to a virtual CPU, the optimization associates each thread with a thread task, which can be in either running or pending state. A running task has a corresponding virtual CPU (OS thread) while a pending task does not. Therefore, though running tasks should generally be no more than physical CPUs, the total number of running and pending tasks can be larger than the number of physical CPUs. After a running task exits (its speculative thread commits or rolls back), if there is a pending task, the pending task is assigned its OS thread and becomes a running task, otherwise its OS thread is returned to the underlying threading implementation. Though the number of tasks N is larger than the number of virtual CPUs P, we note that N is a bounded value with respect to P, in particular, $N < 2P$. This is because each speculative thread needs to create at most one pending task. Moreover, if using an in-order forking model, then only the most-speculative thread needs to create a pending task, and thus $N = P + 1$.

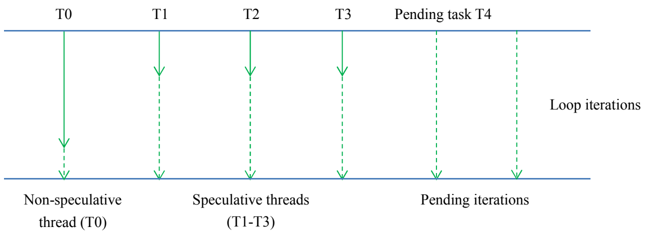


Figure 7. Thread Task Optimization - Normal Execution

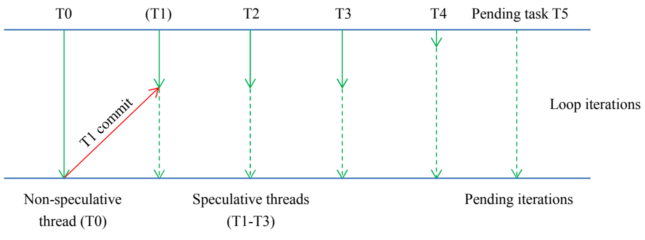


Figure 8. Thread Task Optimization - Thread Joining

The thread task optimization for the example of Figure 6 is demonstrated in Figures 7 and 8. After threads T1, T2 and T3 are in-order speculated, there are no available OS threads, and therefore T3 speculates a pending thread task T4. After T1 commits, T4 is assigned the OS thread of T1 and scheduled to run on a physical CPU. T4 then reaches a fork point and speculates a new pending thread task T5. In implementation, T5 and T1 can share the same rank and thread status and task data.

To rollback and restart the speculative threads for the readonly-page optimization and the buffering integration mechanism, such as the example shown in Figure 5, we have different design choices. One is to rollback all speculative threads/tasks. When the non-speculative thread reaches a fork point again, a child thread is then forked to continue speculative parallel execution. An example of this design is illustrated in Figure 9. The non-speculative thread T0 in-order speculates threads T1, T2, T3 and the pending task T4. Then a thread writes a readonly page and initiates speculative threads/tasks rollback/restart, which causes all speculative threads/tasks to rollback. This design has the drawback of reduced parallel thread work coverage: after the speculative thread tasks are rolled back, the non-speculative thread has to complete the rest of its iteration before speculating a child thread again, resulting in unnecessarily longer program execution time.

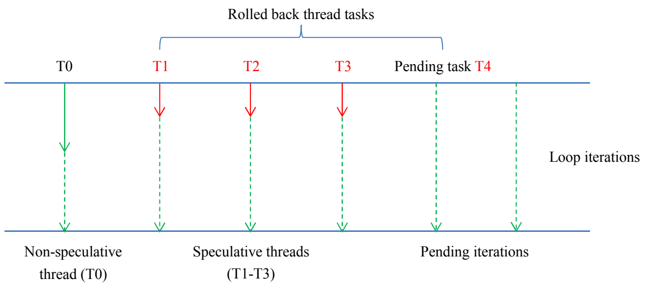


Figure 9. Speculative Threads/Tasks Rollback/Restart without Thread Stopping Optimization

We propose the thread stopping optimization to resolve this issue. When the speculative threads need to be restarted, the non-speculative thread initiates speculative threads/tasks stop/rollback/-restart, which stops its direct children, and then indirect children are cascadingly rolled back. Then after rolling back the eager buffering and resetting the variable page types, the non-speculative thread restarts the stopped child thread tasks.

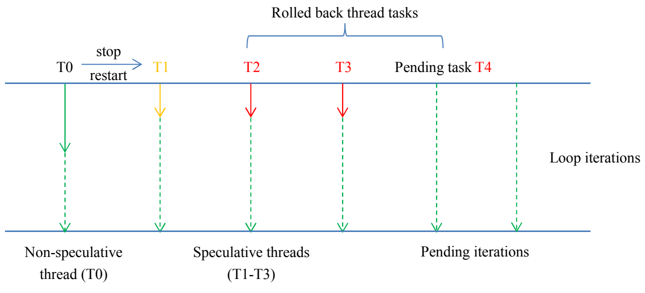


Figure 10. Speculative Threads/Tasks Stop/Rollback/Restart with Thread Stopping Optimization

The example of Figure 9 with the thread stopping optimization is demonstrated in Figure 10. When the speculative threads/tasks T1 to T4 need to restart, the speculative thread T1 is stopped, which in turn cascadingly rolls back T2, T3 and T4. Then as there are no speculative threads running, the non-speculative thread T0 can maintain the buffering metadata and global main memory. Afterwards, it restarts the stopped speculative thread task T1, which in turn re-speculates child threads/tasks T2, T3 and T4.

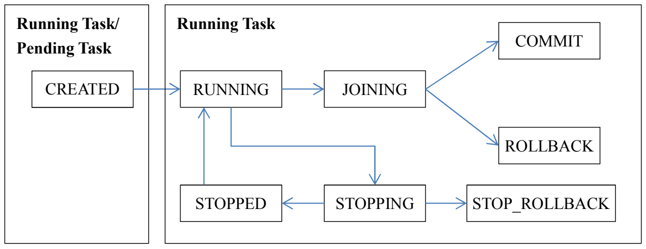


Figure 11. State Transition of Thread Stopping Optimization

The state transition of the MUTLS framework design with the thread stopping optimization is illustrated in Figure 11. When a speculative thread has an invalid buffering memory access such as writing a readonly page or reading/writing an independent WORD and needs to restart, it waits for its state to be STOPPING. When the non-speculative thread has an invalid buffering memory access or finds one of a speculative thread in the `check_valid` call as was discussed for Figure 5, it sets the states of the speculative threads/tasks to STOPPING. If a speculative thread finds its state is STOPPING during thread joining with the non-speculative thread or during waiting after an invalid buffering memory access, there are two cases, depending on whether the speculative thread is a direct child of the non-speculative thread: if it is, it sets its state to STOPPED and exits; otherwise, it transits to STOP_ROLLBACK and calls the `rollback_self_sp` method of Figure 5 to rollback itself. The reason that indirect child threads do not transit to the ROLLBACK state is that, instead of calling `rollback_self_sp` of Figure 5, a thread normally rolling back from the ROLLBACK state calls the `rollback_page_sp` method for all its accessed pages, as was discussed for Figure 4. After the non-speculative thread rolls back the eager buffering and resets the buffering metadata, it resets the states of the stopped direct child threads to RUNNING and restarts the threads.

Benchmark	Description	Problem Size	Language	Source
lavaMD	3D hierarchical particle simulation	10 boxes in each dimension	C	Rodinia
streamcluster	online stream data clustering	65536 points, 1000 clusters	C++	Rodinia
kmeans	k-means clustering	494020 points	C	Rodinia
srad	speckle reducing anisotropic diffusion imaging	609×590 image	C	Rodinia
cfd	3D fluid computational dynamics	97046 elements	C++	Rodinia
sparsematmul	sparse matrix times vector	$2M \times 2M$ matrix, 100M non-zeros	C	SciMark
smallpt	path tracing global illumination rendering	800×600 image, 4 ray samples	C++	smallpt
bwaves	blast waves simulation	train run	Fortran	SPEC CPU2006
fft	recursive Fast Fourier Transform	2^{20} doubles	C	MUTLS

Table 1. Benchmarks

5. Experiments

The MUTLS system is implemented in llvm-3.5. Performance is evaluated on a 4x16-core AMD Opteron 6274 machine with 64GB memory. We use SSE4 instructions for SIMD acceleration. The benchmarks are listed in Table 1, and were selected because they are memory intensive workload applied in a variety of areas and expose significant opportunities for parallelism. They have no memory dependencies. We use train run for bwaves because it takes a long time to run all versions for the ref run data set. As more parallel thread work generally requires more validation/commit (V/C) time, we use more cores for parallelized V/C as more cores are used for speculative parallelization. For the experiment, we use 0, 0, 1, 2, 3, 5, 7, 7 and 7 dedicated parallelized V/C cores when there are 1, 2, 4, 8, 16, 32, 48, 56 and 64 available CPU cores, respectively.

We first experiment with the lazy buffering. The speedups to the original (non-MUTLS) sequential program are shown in Figure 12. The *mutls* version is the MUTLS buffering [3]. The *simd*, *pvc* and *ro* mean the SIMD acceleration, parallelized V/C and readonly-page optimizations, respectively.

It can be seen that the readonly-page optimization is highly effective and efficient, and significantly improves the performance of all benchmarks except fft, which has no large readonly variables. The SIMD acceleration optimization improves performance considerably for srad, bwaves and fft, and moderately for others. The parallelized V/C optimization significantly benefits more V/C intensive benchmarks such as streamcluster, srad and cfd, sparsematmul, bwaves and fft, as a result of reduced critical path delay, while degrading the speedups of data-reuse-intensive benchmarks such as lavaMD and smallpt, which demonstrates the trade-off between the speculative execution time and the validation/commit time. We can see that generally few or no dedicated parallelized V/C cores are needed on machines with no more than 8 cores.

We then experiment with the eager buffering and present the speedups in Figure 13. The *eager-nolazy* version uses the eager buffering and rolls back threads with RAW, WAR and WAW dependencies. The *simd-eager* tries to use the eager buffering for independent global variables and falls back to the *simd* lazy buffering for dependent ones, and *simd-eager-ro* also enables the readonly-page optimization. We also show the speedups of the *simd-pvc-ro* version for comparison with the optimized lazy buffering.

We can see that the readonly-page optimization also significantly benefits the eager buffering, demonstrating its low overhead. The eager versions generally have higher speedups with more cores than the lazy versions as a result of the higher scalability of the eager buffering, yet lower with few cores due to the eager buffering slowing down the non-speculative thread. The plunge of the streamcluster *eager-nolazy* version at 48 and 56 cores is because the benchmark uses a boolean array while we track dependency of the eager buffering using 32-bit WORD, as was discussed in section 4.2, and thus causes rollbacks due to false dependencies. On the other hand, this benchmark demonstrates the advantage of accurate dependency tracking of the lazy page-table buffering discussed in section 4.1. For the bwaves benchmark, the eager buffering causes rollback due to memory access to the stack variables of the non-speculative thread. We cannot apply the eager buffering to the non-speculative stack, since speculative threads may access spilled register variables and/or stack pointers, and thus misspeculation may corrupt the non-speculative stack. Since the eager buffering requires the use of in-order forking model, the lazy versions achieve much higher speedups than the eager versions for fft, resulting from the mixed forking model exploiting more parallelism from the tree-form recursion benchmark [3].

The performance results of the adaptive buffering selection heuristics and thread stopping optimization are presented in Figure 14. We scale the speedups to the *simd-eager-ro* version, and thus these versions have higher/lower speedups than *simd-eager-ro* if the speedup ratios are larger/less than 1, and higher/lower scalability if the curves go upward/downward with the number of cores. The *heuristics* version enables adaptive buffering selection heuristics for the *simd-eager-ro* version. We also show the *lazy-ro* results to compare the *heuristics* version with the corresponding lazy buffering version. The *nostopping* version disables the thread stopping optimization for the *simd-eager-ro* version.

We can observe that the adaptive buffering selection heuristics help to select the appropriate buffering. For srad and cfd, the heuristics select the lazy buffering from 1 to 16 and 1 to 4 cores, respectively, for its lower overhead on the non-speculative thread, and the eager buffering for more cores to benefit from its higher scalability, resulting in optimal solutions for different environments. The *heuristics* version also shows good overall improvement. It is faster than the *simd-eager-ro* version for streamcluster, srad, cfd and bwaves, and faster than the *simd-pvc-ro* version for most benchmarks with more than 8 cores. However, for some benchmarks such as lavaMD, kmeans and smallpt, if eager buffering is selected, the heuristics may degrade the speedups due to extra rollbacks. We also see that there are scenarios where the buffering integration overhead could not be reduced by the heuristics, due to factors such as i-cache miss, branch prediction and more control flow/data access disabling compiler optimization. For example, while the heuristics select the lazy buffering for streamcluster and sparsematmul, the performance is still similar to that of the *simd-eager-ro* version.

The thread stopping optimization is effective for benchmarks with shared and independent variables in loop speculative regions. With more than two cores, the optimization significantly improves the speedups of lavaMD, srad, cfd, sparsematmul and bwaves, as a result of more parallel thread work coverage. On the other hand, using two cores the speedups of the srad and cfd benchmarks are significantly higher without the thread stopping optimization; this is because of the sequential region optimization of section 4.3, for which the non-speculative thread directly accesses the main memory without the eager buffering.

Although other software-TLS systems use different hardware and benchmarks, and are non-trivial to port, a rough comparison can be made to see how our results compare with the performance of other, pure eager designs. The speedup ratios of the *simd-eager-ro* version to the OpenMP manual parallel version are presented

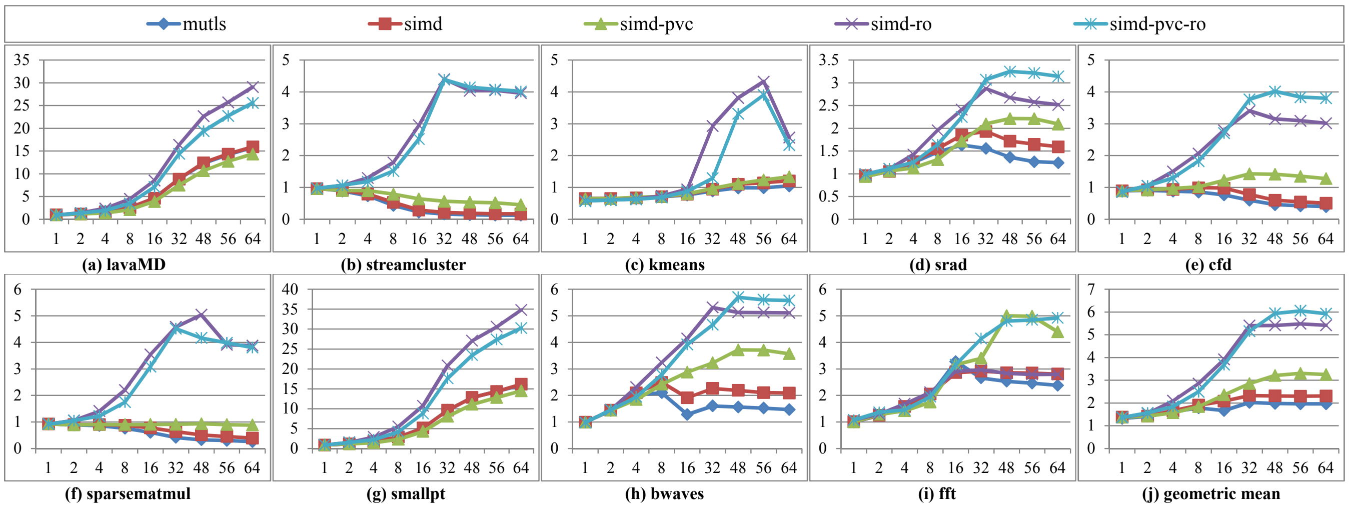


Figure 12. Speedup versus number of cores; higher is better.

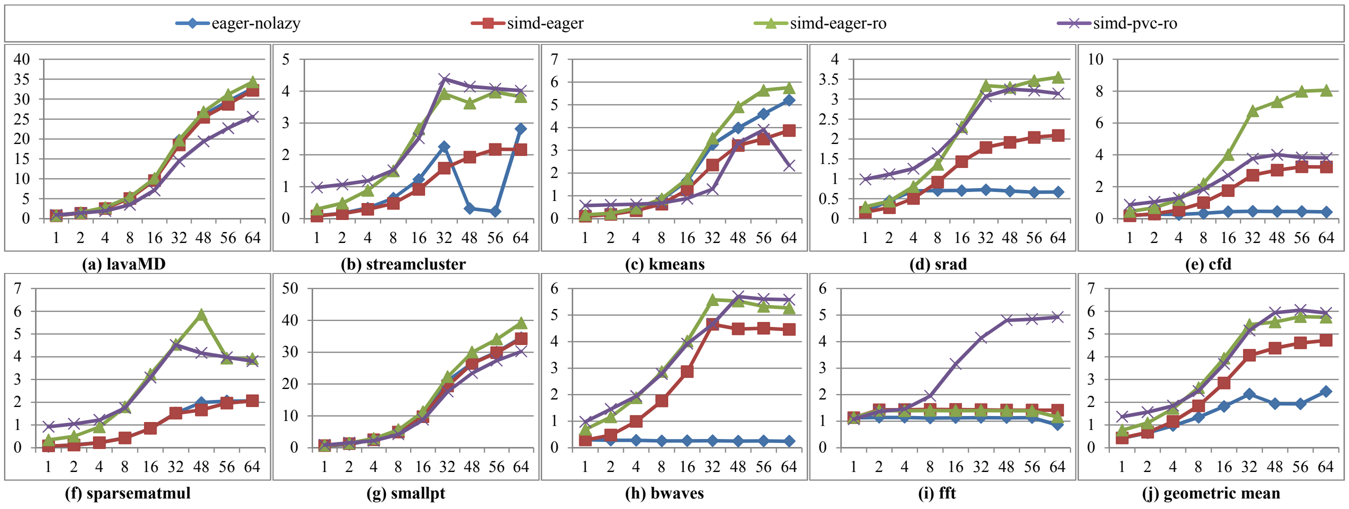


Figure 13. Speedup versus number of cores; higher is better.

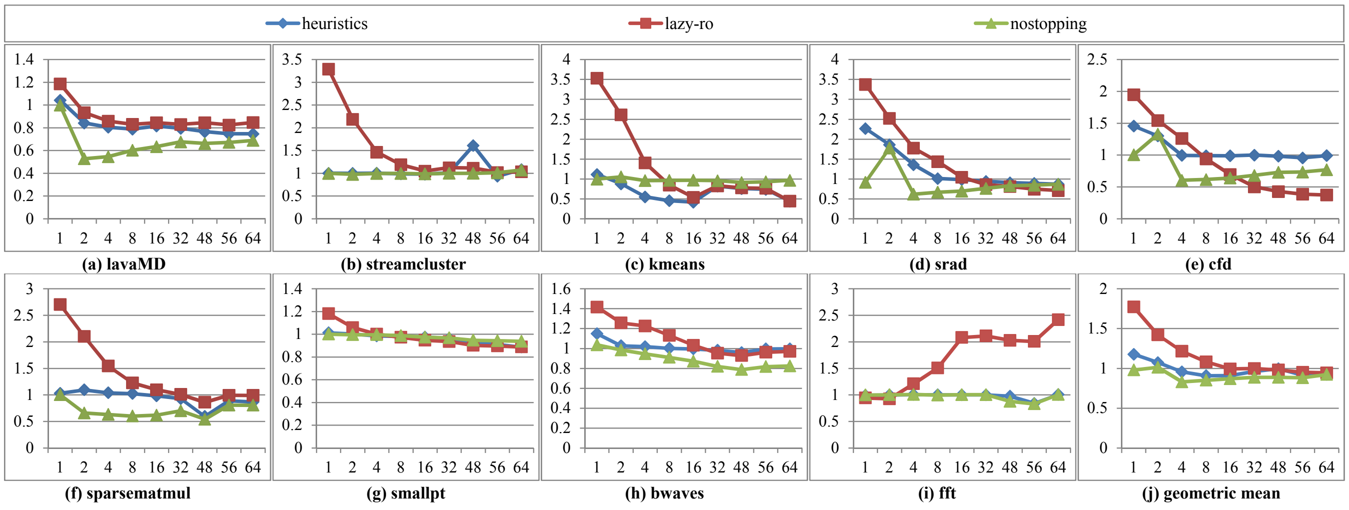


Figure 14. Speedup ratio versus number of cores, scaled to the *simd-eager-ro* version; higher is better.

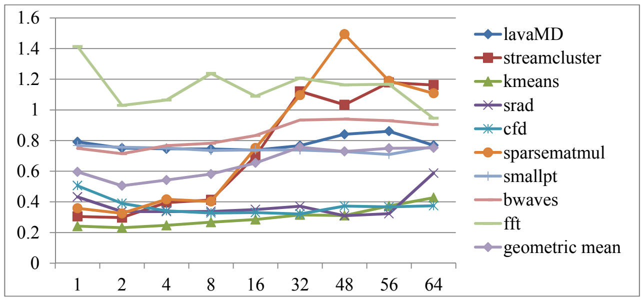


Figure 15. Speedup ratio versus number of cores, scaled to the OpenMP version; higher is better.

in Figure 15, which also shows the relative memory buffering overhead since OpenMP has no buffering. On an 8-core machine SpLIP achieves 98.5% and 71.7% of the performance of manually parallelized versions of sparse matrix multiplication and barnes hut (related to sparsematmul and lavaMD) [13], whereas our *simd-eager-ro* version reaches 110.8% and 76.9% of the OpenMP version, although requiring 32 cores to converge. We do better with respect to MiniTLS, which achieves speedups of 3.7 and 4.8 respectively on the same benchmarks using 32 cores [23], while *simd-eager-ro* has 4.5 and 19.7 speedup on 32 cores, and can utilize 64 cores to achieve speedup of 3.9 and 34.4. Adaptive MUTLS of course also handles benchmarks such as bwaves that respond better to lazy approaches, and has the additional benefit of being language and architecture neutral, while SpLIP and MiniTLS only support a specific language/runtime environment (C++ and Java, respectively).

6. Conclusions and Future Work

Efficient memory management is critical to the design of effective software approaches to thread-level speculation, with the competing buffering strategies used to either enforce isolation or to preserve undo information having different costs and potential benefits. We initially approached the problem as one of establishing which technique is better, developing highly optimized, but separate implementations of the buffering approaches. Both techniques improve performance, and both benefit from further optimizations to identify readonly data and so reduce buffering costs, but it depends very much on the benchmark and resource limits. By combining the techniques and performing an adaptive, runtime selection of the buffering mechanism we are thus able to demonstrate a design that gains the benefits of both, with a more general application that accommodates different benchmarks and numbers of available cores.

Future work involves further tuning the buffering integration mechanism—our adaptive heuristics are effective, but could perhaps be improved by maintaining different buffering integration data for different fork points, which should reduce the rollback time ratio for programs with iterations containing different speculative regions such as bwaves. More precise, and ideally ahead-of-time identification of independent or readonly variables may also be possible through static analysis, profilers, and/or feedback logs. We are also interested in exploiting common hardware accelerators such as GPU and hardware transactional memory (HTM), as a means of further alleviating overhead without sacrificing the advantages of software TLS in applying to existing, commodity hardware.

Acknowledgments

This work supported by the Natural Sciences and Engineering Research Council of Canada, Application ID #249902.

References

[1] Z. Cao. MUTLS (mixed model universal software thread-level speculation). `http://www.sable.mcgill.ca/~zcao7/mutls`, 2013.

[2] Z. Cao and C. Verbrugge. Adaptive fork-heuristics for software thread-level speculation. In *PPAM'13: 10th International Conference on Parallel Processing and Applied Mathematics*, pages 523–533, 2013.

[3] Z. Cao and C. Verbrugge. Mixed model universal software thread-level speculation. In *ICPP'13*, pages 651–660, 2013.

[4] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA'03*, pages 434–446, June 2003.

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

[6] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI'04*, pages 71–81, June 2004.

[7] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2(3):247–279, Sept. 2005.

[8] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *PPoPP'07*, pages 215–225, Mar. 2007.

[9] M. Lupon, G. Magklis, and A. Gonzalez. A dynamically adaptable hardware transactional memory. *MICRO'43*, pages 27–38, 2010.

[10] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Distributed Computing*, pages 354–368. Springer, 2005.

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

[12] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE'08: 1st International Workshop on Multicore Software Engineering*, pages 23–32, May 2008.

[13] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA'09*, pages 223–232, Aug. 2009.

[14] M. Payer and T. R. Gross. Performance evaluation of adaptivity in software transactional memory. In *ISPASS'11*, pages 165–174, 2011.

[15] C. J. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05*, volume 4339, pages 304–318, 2005.

[16] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI'05*, pages 269–279, June 2005.

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

[18] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC'06: Proceedings of the 20th international conference on Distributed Computing*, pages 284–298. Springer, 2006.

[19] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3:1–28, Oct. 2001.

[20] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, Aug. 2005.

[21] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO'08*, pages 330–341. IEEE Computer Society, 2008.

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

[23] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Lujan. Optimizing software runtime systems for speculative parallelization. *ACM Transactions on Architecture and Code Optimization*, 9(4):39:1–39:27, Jan. 2013.

[24] L. Zhao, W. Choi, and J. Draper. Sel-tm: Selective eager-lazy management for improved concurrency in transactional memory. In *IPDPS'12*, pages 95–106, 2012.