Fast Memory Snapshot for Concurrent Programming without Synchronization

Jaewoong Chung
Research and Advanced Development Lab
AMD Corporation
jaewoong.chung@amd.com

Woongki Baek, Christos Kozyrakis
Computer Systems Lab
Stanford University
{wkbaek,kozyraki}@stanford.edu

ABSTRACT

The industry-wide turn toward chip-multiprocessors (CMPs) provides an increasing amount of parallel resources for commodity systems. However, it is still difficult to harness the available parallelism in user applications and system software code.

We propose MShot, a hardware-assisted memory snapshot for concurrent programming without synchronization code. It supports atomic multi-word read operations on a large dataset. Since modern processors support atomic access only to a single word, programmers should add synchronization code to process a multi-word dataset concurrently in multithreading environment. With snapshot, programmers read the dataset atomically and process the snapshot image without synchronization code. We implement MShot using hardware resources for transactional memory and reduce the storage overhead from 2.98% to 0.07%. To demonstrate the usefulness of fast snapshot, we use MShot to implement concurrent versions of garbage collection and call-path profiling. Without the need for synchronization code, MShot allows such system services to run in parallel with user applications on spare cores in CMP systems. As a result, the overhead of these services is minimized, approaching that of an ideal implementation.

Categories and Subject Descriptors

B.3.0 [**Hardware**]: MEMORY STRUCTURES—*General*

General Terms

Design

Keywords

Snapshot, Transactional Memory

1. INTRODUCTION

Chip-multiprocessors (CMPs) bring abundant parallelism to commodity systems. However, the lack of concurrency in software prevents programmers from fully exploiting the additional hardware cores. Ideally, sequential code could be easily changed to use multiple cores. However, parallelization is not trivial in practice because programmers must deal with the complications of concurrency management.

To alleviate this problem, it is important to develop architectural tools that help programmers to exploit parallelism. In search of such tools, we find *memory snapshot* particularly useful for improving the concurrency of software systems [1, 2, 15, 22]. The key benefit of snapshot is to support atomic multi-word read operations that produce a consistent view on a large dataset. Since modern processors support atomic memory access only to a single word, programmers have to deal with complex synchronization issues to process a multiple-word dataset concurrently in multithreading environment. With snapshot, concurrent programs read a large dataset atomically and work with a consistent snapshot image of the dataset without synchronization code. The basic concept of snapshots has been used widely in databases, file systems, and reliable storage [11, 21, 33]. While several algorithms for memory snapshot have been proposed, their applicability for performance optimizations is limited due to high runtime overhead since they rely on pure software techniques [1, 2]. More sophisticated software implementations allow for additional gains at the cost of algorithmic complexity for applications [15, 22].

This paper proposes *MShot*, hardware-assisted memory snapshot for concurrent programming without synchronization code. MShot provides a fast memory snapshot with hardware acceleration. Programmers use a snapshot to read a multi-word dataset atomically and to process it concurrently in multithreading environment. The snapshot image is isolated from further memory updates, shared by multiple threads, and accessed with normal load/store instructions. MShot supports multiple memory snapshots of arbitrary lifetime that consist of multiple disjoint memory regions. A fast memory snapshot is beneficial for the software systems in need of atomic multi-word read operations such as fast concurrent backup, checkpointing, debugging parallel programs, concurrent garbage collection, dynamic profilers, fast copy-on-write, and in-memory databases.

We implement MShot using hardware resources available in transactional memory (TM). TM executes a group of instructions in an atomic and isolated manner [8, 17, 19, 24]. The opportunity for sharing resources between MShot and hardware TM is understood intuitively since both systems support some sort of atomic execution. The key idea is to use the cache as a buffer for snapshot data and additional bits per cache line for snapshot metadata like a cache-based hardware TM system does for transactional data and metadata. Due to resource sharing with hardware TM, the storage overhead of MShot is reduced from 2.98% to 0.07%.

To evaluate MShot, we prototyped two snapshot-based systems: garbage collection (GC) and dynamic profiling. Taking advantage

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

of the fact that "once garbage, always garbage," the snapshot-based GC takes a memory snapshot and performs collection without interfering with the application (mutator) threads running in parallel. The snapshot-based profiler takes a snapshot of the stack for call-path profiling. Reading data structures atomically and efficiently without synchronization code, these systems run the tasks concurrently with applications and add only a negligible runtime overhead to the applications.

The rest of the paper is organized as follows. Section 2 presents the motivation for this work. Section 3 explains the definition, programming interface, and hardware implementation of fast memory snapshot. Section 4 explains MShot implementation with TM hardware resources. Section 5 presents the quantitative evaluation. Section 6 discusses related work, and Section 7 concludes the paper.

2. MOTIVATION

2.1 Motivating Example

Modern processors support atomic read and write operations on single-word data [1,2]. However, if multiple words need to be read atomically while some of the words are written concurrently, systems do not provide a consistent view of the words for the read operation. Let's consider an example that mimics dynamic memory profiling where a profiling thread traverses an object reference graph and an application thread changes the graph concurrently. In Figure 1(a), the profiling thread has visited node A, determined that node B was the only child node, and reads node B while the application thread has detached node D from node C and is attaching it to node A. Since node A has already been visited, the profiling thread fails to visit node D, a new child node of node A, unobserved before. The problem is that the profiling thread cannot read the whole graph atomically and may miss visiting some live objects due to concurrent mutation by the application thread.

There are several solutions to address the problem. The profiling thread can grab a global lock for the graph, but in doing so it loses concurrency. The two-phase locking in database literature grabs per-node locks gradually and releases all locks when the graph traversal completes. It allows for more parallelism, but introduces lock acquisition overhead and may cause deadlocks when the profiling thread and the application thread traverse the same link in the opposite directions as shown in Figure 1(b). Transactional memory (TM) [17, 19, 24] guarantees the atomic and isolated execution of the instructions in a transaction. With TM, the profiling thread can read the whole graph atomically by enclosing all graph access instructions within a transaction. However, the transaction is likely to be much longer than usually expected [10] and may suffer from frequent restarts due to constant conflicts with the application thread as shown in Figure 1(c). Virtual memory protection [5] and Dynamic Binary Translation(DBT) [37] can be used to save old values of the graph for the profiling thread. However, virtual memory protection suffers from the page fault exception overhead and DBT from the instruction instrumentation overhead.

There have been developed algorithms specific to concurrent graph traversal such as tricolor marking [18] where three colors (i.e., white, gray, and black) are used to present the traversal status of each node. This scheme solves the problem by changing the color of node A from black (i.e., traversal done) to gray (i.e., traversal ongoing) when node D is attached to node A as shown in Figure 1(d). Since it takes advantage of knowledge on the data structure such as pointers to child nodes and the number of child nodes, this scheme is not easily generalized for atomic multi-word read operations.

2.2 Hardware-assisted Memory Snapshot

Memory snapshot provides support for atomic multi-word read operations [1,2]. A "snapshot" of m memory elements is created to provide a consistent view for p processors. Then, the processors are allowed to execute two operations: *update* to write a memory element in the snapshot and *scan* to read memory elements. The scan operation is an atomic read operation on the memory elements and produces a consistent copy of them at the moment the scan is executed. Memory snapshot solves the race problem easily without synchronization as shown in Figure 1(e). A memory snapshot is taken (i.e., scan) on all nodes of the graph. The profiling thread reads the pointer from node C to node D in the snapshot image while the application thread modifies the pointer in the up-to-date image. Unfortunately, despite the great potential as a programming primitive for concurrent programming, memory snapshots have been implemented in software and have performance issues such as O(mp) update time [1,2] or O(m) scan time [15,22], which prevent them from being adopted in a wide range of applications.

We propose fast memory snapshot with hardware assistance for easy concurrent programming without synchronization code. It accelerates snapshot operations with hardware resources to provide O(1) update (as fast as a single memory write operation) and O(p) scan in O(m) space. The O(1) update time enables application threads to write to the up-to-date image without performance degradation. The O(p) scan time allows snapshots to be used for large datasets in performance-oriented programs. We use the cache as a buffer for snapshot data and additional bits per cache line for snapshot metadata. A consistent snapshot image is taken with interprocess communication and maintained with cache coherence protocol support. Using hardware acceleration, memory snapshot can be applied to performance-oriented software system, including the following:

- **Fast Concurrent Checkpoint**: Process state is checkpointed for backward recovery with checkpoint in fault tolerance mechanisms [5] or for guest OS migration in virtual machines [36]. The fast memory snapshot is used to create the backup memory image concurrently without slowing down the applications (due to O(1) update time). Logging threads run in parallel with application threads to log the checkpointed image into disks.

- **Concurrent Garbage Collection**: Concurrent garbage collection typically incorporates sophisticated algorithms to deal with races between mutators and collectors and increases code management cost [12]. With fast memory snapshots, concurrent garbage collection can be as simple as a stop-the-world garbage collector by taking a snapshot of part of the heap and collecting garbage from the snapshot image concurrently.

- **Concurrent Memory Profiling**: As shown in the example in the previous section, concurrent memory profilers [27,28] benefit from fast memory snapshot by traversing a consistent object reference graph in the snapshot image.

- **Concurrent Call-Path Profiling**: Just-In-Time (JIT) compilers in Java virtual machines and the C# runtime system optimize the application code by finding hot execution paths with call-path profiling [13]. With fast memory snapshot, compilers take a snapshot of a thread stack periodically and analyze the stack in parallel with application threads.

- **Fast Copy-On-Write (COW)**: COW is used for shared virtual-to-physical page mappings in fast *fork()* [32] and for disk

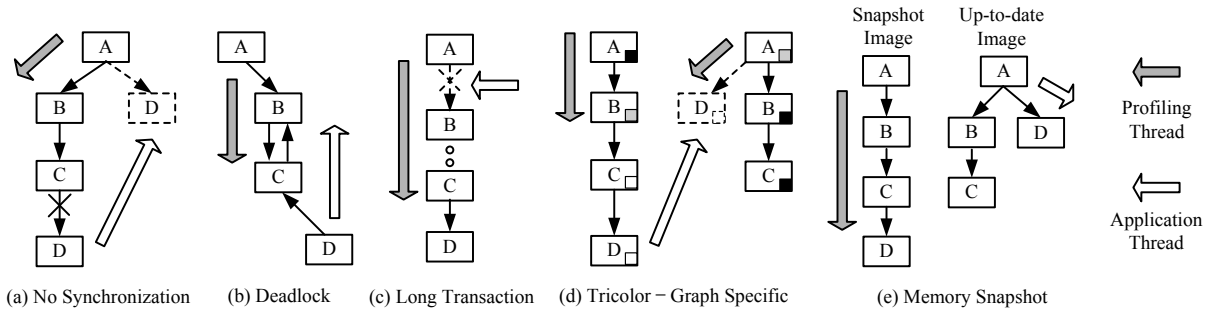

Figure 1: Concurrency Issues and Synchronization Schemes.

block sharing between virtual machines [34]. The performance problem with COW is that the thread modifying shared data is stalled until a safe copy of the data is made. With fast memory snapshot, the thread takes a snapshot of the data and modifies the data without being stalled for the copy to be made. The copy is made in parallel from the snapshot image.

- **In-memory Database**: Unlike traditional database systems, in-memory database systems achieve good performance by loading the whole data schema into main memory [14]. The snapshot image on the in-memory data schema can be used to generate reports on data usage, to replicate the data in a database cluster, and to support the isolation-level necessary for database transactions.

In summary, fast memory snapshot is useful for applications that need atomic multi-word read operations to obtain a recent and consistent view of various data structures.

3. FAST MEMORY SNAPSHOT DESIGN

We propose MShot, a hardware-assisted memory snapshot for atomic multi-word read operations on arbitrary datasets. We first present the definition and programming interface for fast memory snapshot and then explain the hardware mechanism.

3.1 Definition and Interface

MShot provides a snapshot of the memory image at a certain point in time. Multiple disjoint address regions can be part of a snapshot. Once a snapshot is taken on these regions (i.e., equivalent to a scan operation of software snapshots), MShot maintains two memory images: the master image with the up-to-date data and the snapshot image. To read from the snapshot image, threads first join the snapshot. Any number of threads can join the snapshot. Until the user threads leave the snapshot, normal read operations to the snapshot regions return the snapshot data. The separation of taking a snapshot from joining the snapshot makes it possible to take a snapshot in a performance critical section and analyze the image later using additional cores. A snapshot is destroyed after all user threads have left the snapshot. A snapshot image in MShot is read-only like software snapshots [1, 2, 22]. MShot also captures the associated core register values at the moment the snapshot is taken. This is useful for analyzing the image later on. There can be multiple snapshots active at any moment. To simplify the hardware requirements, MShot does not allow overlapping snapshot regions and only supports cache-line granularity in specifying regions.

Table 1 shows the programming interface of MShot to take, join, leave, and destroy a snapshot. A simple data structure called *snapshot_info* shown in Figure 2 is used to set up a snapshot. A unique snapshot ID (SID) is assigned to a newly taken snapshot.

3.2 Implementation

There are three key software and hardware components in MShot implementation as shown in Figure 3.

The **Snapshot Information Table (SIT)** shown in Figure 3(a) is a doubly-indexed hash table for managing all snapshot information. This software structure is indexed either by SID or by virtual address. SIT entries, called *Snapshot Information Blocks (SIB)*, are similar to the *snapshot_info* structures passed to *take_snapshot()*. *join_list* is a linked list of the threads that joined a snapshot. MShot increments a simple counter atomically to generate a unique SID.

The **Snapshot information Look-aside Buffer (SLB)** shown in Figure 3(b) is a small 64-entry hardware cache for accelerating the retrieval of snapshot information from the SIT. Each entry encodes information about a snapshot region. The SLB is accessed in parallel with the TLB. A matching SLB entry returns two fields: the SID to which the snapshot region belongs and the *J (join)* bit indicating if the current thread has joined the snapshot. The SLB maintains an *OV*(overflow) bit to indicate that there is a SLB entry evicted due to capacity issues. If the bit is not set, an SLB miss is ignored since there is no snapshot region associated with the memory address. If the bit is set, the miss is handled by a software refill handler that accesses the SIT. To support the software handler, two SLB instructions are added: *SLBI* to invalidate SLB entries and *SLBLD* to load them. SLBI and SLBLD are similar to TLBI and TLDLD in PowerPC [30].

Two Snapshot Metadata bits and SID bits are added per cache line for data versioning. *MS (Modified since Snapshot)* bit is set when a cache line in a snapshot region has been modified since the snapshot is taken. *RS (Read from Snapshot)* bit is set when a cache line in a snapshot region is refilled with old data from the snapshot image. If both bits are not set for a cache line, it implies that the master image and the snapshot image have the same data for the line. A 6-bit SID is set when either the MS bit or RS bit is set to indicate the snapshot for which the bits are set. The SID bits are set to 0 for the cache lines unrelated to a snapshot.

Take_snapshot() initiates the process of taking a snapshot. An SIB is created for the snapshot by copying *snapshot_info* to the block and inserting it into the SIT. MShot ensures that all cores are aware of the snapshot information before the call returns. This guarantees any write to the snapshot after the snapshot is taken triggers data versioning to build the snapshot image gradually, avoiding O(m) scan time. The snapshot information is exchanged by using the three-way handshake shown in Figure 4. First, snapshot request messages are sent via inter-core signals to the other threads, invoking snapshot signal handlers. The handlers copy the saved register values of the threads to a pre-allocated data structure pointed to by the *saved_regs* field of the *snapshot_info* argument, and load the snapshot information into the SLB. Next, the handlers send re-

Group	Method	Function
Snapshot Control	take_snapshot (snapshot_info*)	Take a snapshot on the address regions specified by snapshot_info.snapshot_regions. New snapshot id (SID) is set at snapshot_info.SID. Saved register values are pointed to by snapshot_info.saved_regs.
	destroy_snapshot (SID)	Destroy the snapshot of SID.
Snapshot Sharing	join_snapshot (SID)	Start using the snapshot of SID.
	leave_snapshot (SID)	Stop using the snapshot of SID.

Table 1: MShot software interface.

```
snapshot_info {
  short SID;                        // unique snapshot id
  long[][2]* snapshot_regions;      // array of (start address, end address) pairs
  void* saved_regs;                 // saved register values
  short TID;                        // optional. Thread id to isolate selectively
}
```

Figure 2: `snapshot_info` data structure used for *take_snapshot()*.

sponse messages back to the thread that initiated the snapshot and wait for resume messages. On receiving the response messages from all the handlers, the initiating thread sends a resume message to terminate the handlers and resume application threads. This process has O(p) complexity.

Join_snapshot() updates the SIB to remember that the thread is a new *user* of the snapshot. The corresponding SLB entry is reloaded to set the J bit for the thread.

Cache operations with the snapshot metadata bits are summarized in Table 2. If a user thread writes to a snapshot, an exception is triggered since the user thread is only allowed to read from the snapshot. The SLB detects this case. A read from the snapshot is a hit if there is a cache line with a matching address tag and its MS bit is 0. This indicates that the line has not been modified since the snapshot was taken. If it is a miss, the SID bit and J bit are piggybacked to the refill request to indicate that the refill should come from the snapshot image. On receiving the refill request, other caches search for a cache line with matching address tag whose MS bit is 0. If the MS bit of the cache line with matching address is 1, the MS bit is attached to the response message to notify the requester that the master image has deviated from the snapshot for that address. Finding no matching cache line in other caches, the request is sent to main memory and the data is refilled. When refilling the cache line, we set the RS bit if the MS bit is piggy-backed. The RS bit is used later when destroying the snapshot to see if the line is to be invalidated.

A read from or a write to the snapshot by a thread not using the snapshot (i.e., J bit is 0) is a hit if there is a cache line with a matching address tag and the RS bit is 0. This indicates that the line does not contain the old data from a snapshot. If it is a write hit, the MS bit is set to indicate that the line no longer belongs to the snapshot image. This indication is accomplished system-wide by simply setting the MS bit since the line is exclusive to that processor. This makes updates O(1). If the write misses, the MS bit is set after the line is refilled. A cache miss is handled similar to the case of a thread that has joined the snapshot. The SID and J bit are piggy-backed on cache line refill requests. The bits are used by the other caches to find a cache line whose address tag matches and whose RS bit is 0.

Since the cache capacity is limited, a long-lived snapshot may cause cache overflow. Snapshot virtualization mechanism is explained in conjunction with transactional memory in the next section.

Leave_snapshot() is called to stop using the snapshot. The ID of the thread is removed from the SIT and the SLB entries are reloaded to clear the corresponding J bits.

Destroy_snapshot() destroys the snapshot. It starts with invalidating the SLB entries. Cache lines with the SID of the snapshot are invalidated if the RS bit is set. All metadata bits of the snapshot are gang-cleared. It completes by removing the SIB of the snapshot from the SIT.

4. RESOURCE SHARING WITH TRANSACTIONAL MEMORY

Some astute readers would have already noticed the similarity between the MShot implementation and a typical cache-based hardware TM system (HTM). The similarity comes from the fact that they use essentially the same mechanism for data versioning where they both use the cache as a buffer for multiple data versions and add metadata bits per cache line for version information. Expecting that HTMs will be adopted widely in the near future, we present the base hardware TM system assumed in this paper and explain how MShot uses the hardware resources for HTM. At the end of this section, we calculate the hardware cost shared by MShot and HTM.

4.1 Baseline Hardware TM System

Hardware TMs use hardware resources to accelerate transactional execution of a group of instructions [4, 8, 17, 24]. The baseline hardware TM assumed in this paper records transactional loads and stores with two TM metadata bits per cache line: R bit for loads and W bit for stores [17, 24]. They are used to distinguish transactional data and non-transactional data. Transactional data are buffered in the cache [8, 17]. It supports fast context switching of transactional code using transaction IDs per cache line [23, 25].

HTM systems are limited by the capacity of hardware caches. The baseline HTM system uses Page-based TM (PTM) [9] to deal with the cache overflow. PTM supports the features necessary for MShot such as paging of overflowed pages and has a reasonable incremental hardware cost as will be shown in Section 4.3. PTM maintains a software shadow page table with the hardware table access logic located next to the memory controller [9]. When a cache-line with transactional data or metadata is evicted, PTM allocates a new page (shadow page) to store the last committed version of the data and uses the old page (home page) to store the overflowed data. The shadow page table maintains the proper mapping information.

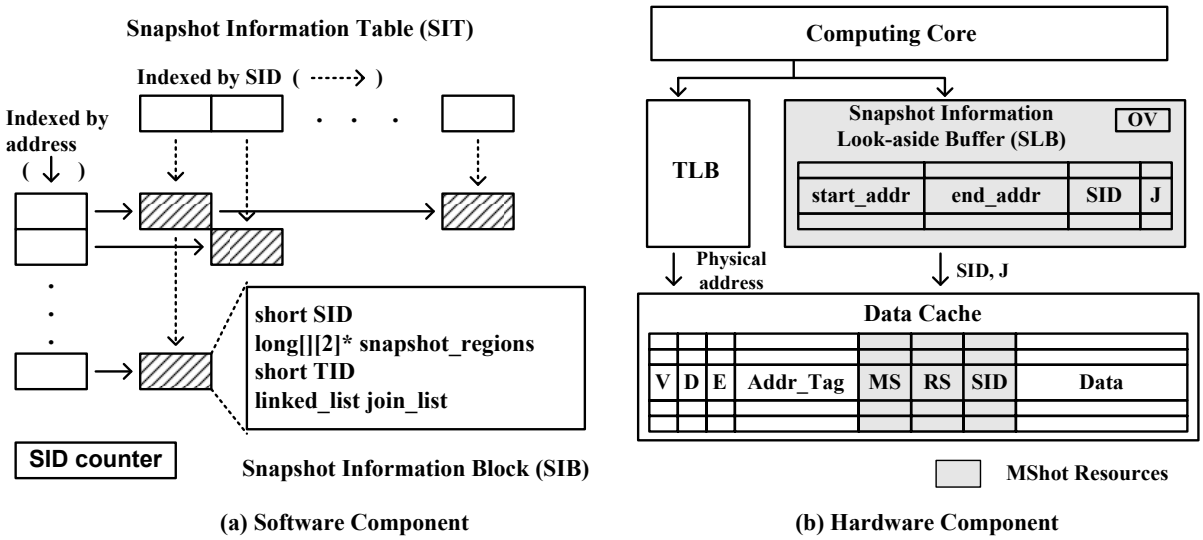

Figure 3: MShot hardware and software structures.

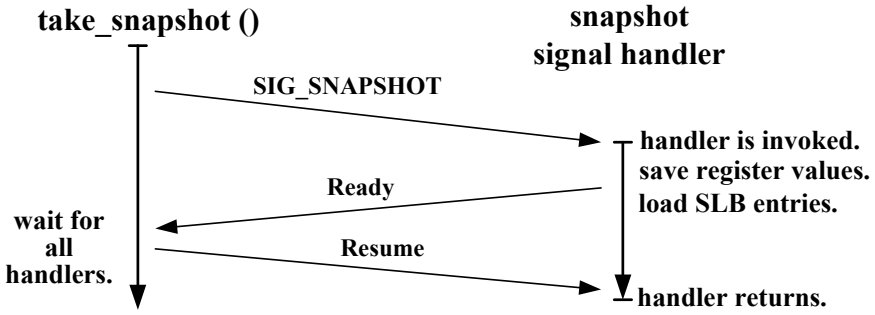

Figure 4: Three-way handshaking for snapshot initialization.

Figure 5 shows the hardware resources of the base HTM system in dark gray.

4.2 MShot using TM Hardware Resources

The key idea to support memory snapshot with HTM resources is to map snapshot read/write operations to transactional read/write operations properly. We map a memory snapshot to a read-only transaction that never aborts until the end of the transaction. The values read from the snapshot image are included in the read-set of the read-only transaction. Write operations to the snapshot regions are handled as transactional stores. The conflicts against the read-only transaction are ignored since the snapshot excludes the new updates. Moreover, regardless of the conflicts, the cache lines with the snapshot data are invalidated eventually by aborting the read-only transaction when the snapshot is destroyed.

To realize this mapping, MShot uses the two TM metadata bits for the two snapshot metadata bits, the transaction ID field for the SID, the shadow page table for overflowed snapshot metadata, and the shadow pages for overflowed snapshot data. Table 3 summarizes the resource mapping between the base HTM and MShot. MShot maps the MS bit to the W bit and uses it to distinguish the master image from the snapshot image. Similarly, the RS bit is mapped to the R bit to mark old data from the snapshot image that is invalidated when the snapshot is destroyed. Additional combinational logic is used for the bit control as explained in Section 3.2. SID is mapped to transaction ID to allow multiple snapshots to share the cache. MShot uses the gang-clear logic to reset the metadata bits when a snapshot is destroyed.

The shadow page table is used to deal with cache overflow of snapshot data. MShot uses the home pages to store the master image and the shadow pages to buffer the snapshot image. When a cache line with an MS bit (i.e., up-to-date data) is evicted, the MS bit and SID of the line are piggy-backed. The shadow page table perceives the SID as the transaction ID and the MS bit as the W bit. The snapshot data of the cache line is copied to the shadow page and the evicted up-to-date data of the master image goes to the home page. As in PTM, shadow pages are from a pool of pre-allocated pages. MShot evicts a cache line with the RS bit (i.e., snapshot data) silently as it is part of the snapshot image in the main memory. To reload an overflowed cache line, the hardware attaches the J bit and SID bits with the refill request. The shadow page table uses the J bit to select if the home page or the shadow page is read. If the bit is 1, the shadow page is read. If not, the home page is read. When a snapshot is destroyed, MShot sends out the transaction commit message with the SID of the snapshot telling the shadow page table that it is the end of the read-only transaction. Then, the shadow page table releases the shadow pages that contain the snapshot image. All in all, Figure 5 shows the hardware resources shared with HTM in dark gray and the MShot-specific hardware components in light gray.

	Write to snapshot	Read from snapshot
Snapshot User Threads	Trigger exception.	Hit if address tag matches and MS bit is not set. Set RS bit and SID when refilled with MS bit piggy-backed.
The Rest	Hit if address tag matches and RS bit is not set. Set MS bit and SID when hit or refilled.	Hit if address tag matches and RS bit is not set.

Table 2: Memory operations with snapshot.

HTM resource	Usage in MShot
Transactional metadata bits per cache line	Snapshot metadata bits per cache line
Transaction ID (TID) per cache line	Snapshot ID (SID) per cache line
Transactional metadata bit gang clear logic	Snapshot metadata bit gang clear logic
Shadow page table in SW, table access logic in HW	Providing access to master/snapshot images
Home/Shadow pages	Containing Master/Snapshot images separately

Table 3: Hardware resource mapping between HTM and MShot.

4.3 Hardware Cost Saving

The hardware cost for the MShot implementation consists of the combinational logic for control and the storage overhead for data and metadata. Since the complexity of the combinational logic is hard to quantify, we focus on the storage overhead in this paper. As a baseline system, we assume a 16-core CMP with 64KB 4-way private L1 cache, 1MB 8-way private L2 cache with 32B line size, and 16GB main memory. This configuration is used for evaluation in Section 5 as well. The storage overhead of the additional bits in cache is about 34 KBytes per core, from (2b for snapshot metadata + 6b for SID) * ((64KB + 1MB)/32B). The storage requirements for SLB per core is about 1 KBytes, from (64b for start address + 64b for end address + 6b for SID + 1b for J bit) * 64 entries. Then, the storage overhead of the SLB and additional bits in the cache for 16 cores is about 560 KBytes. The majority of the PTM hardware cost for MShot is from the 2048-entry shadow page table cache in the hardware table access logic. The shadow page table entry consists of physical page numbers for a home page and a shadow page, a valid bit, and read/write summary vectors. As the physical address length is 34 bits for 16GB main memory, the total storage overhead of the table is about 75 KBytes, from (2*22b for home-/shadow physical page number + 1b for valid bit + 2*(4K/32B)b for read/write summary vectors) * 2048 entries. The storage size of the cache structure in the baseline system is about 20.8 MBytes including bits for physical tags (19 bits for L1 and 16 bits for L2), coherence (3 bits), and ECC (4 Bytes).

Overall, the total storage overhead of the hardware resources used for MShot is 2.98%, from (560KB + 75KB)/20.8MB. However, given the baseline HTM, the additional storage overhead is only 0.07%, from (16 * 1KB SLB)/20.8MB. This shows clearly the benefits of resource sharing between MShot and HTM. If the baseline HTM does not have a PTM, the MShot implementation adds the shadow page table for itself. However, the total storage overhead goes up only to 0.43% from (16KB + 75KB)/20.8MB.

4.4 System Issues

Like the TLB, the SLB must be reloaded on context-switches because it is indexed by virtual addresses. The SLB entries do not need to be saved since the SIT is always more up-to-date than the SLB. On rescheduling, the SLB can be loaded either eagerly or lazily. We use the eager approach in our evaluation.

On paging, MShot flushes out the page from the cache to main memory intentionally. The flush causes PTM to generate the shadow page if needed. After paging out the home page normally, the OS checks if there is an associated shadow page by referencing the shadow page table and swaps the shadow page out, too. The two pages are paged in together when the page is accessed again.

Since our baseline TM system has only two TM metadata bits, user transactions are not supported once the bits are used up by MShot. In order to support user transactions together with snapshots, the baseline TM system needs to support nested transactions with at least two pairs of TM metadata bits per cache line [23] to dedicate a pair to user transactions and the other pair to MShot.

Feature	Description
CPU	2GHz, single-issue, in-order x86 core
SLB	64 entries
L1 Cache	64 KB, 4-way, 32B line, MESI, write-back 1 cycle hit time, private
L2 Cache	1 MB, 8-way, 32B block, MESI, write back 10 cycle latency, private, 8 banks bit vector of sharers
Shadow PT	2048 entries
Memory	4 GB, 100 cycle latency
Interconnect	Tiled network, 32B links, 3 cycles per hop

Table 4: Parameters for the simulated CMP system.

5. EVALUATION

5.1 System and Applications

We implemented MShot on a cycle-accurate simulator and evaluated it with the parameters in Table 4. We used nine applications and one micro benchmark. W3M is a client-side web browser [35]. Pypy is a python interpreter [31]. Gzip is a compression tool [16]. Mpeg2 is a MPEG-2 decoder [3]. Cfrac performs continuous fraction factorization for integers [6]. Nullhttpd is an HTTPD web server [29] . Vacation mimics an e-commerce system [7]. Mp3d and Radix are from SPLASH2 [38]. RBtree is a micro benchmark that adds, searches, and deletes objects into and from a red-black tree in transactions.

We evaluate MShot with garbage collection and call-path profiling. **Snapshot GC** takes advantage of the fact that "once garbage, always garbage." If a garbage object is found in the snapshot image, it is garbage in the master image as well. It takes a snapshot on the memory regions to collect and has collectors find garbage from the snapshot image while application threads modify the master image. **Snapshot call-path profiling** periodically takes a snapshot of the thread stack and walks the stack to obtain information on the call graph of functions in the application code. We compare snapshot GC with parallel GC that stops the world and collects garbage with multiple collectors in parallel [39]. Snapshot call-path profiler

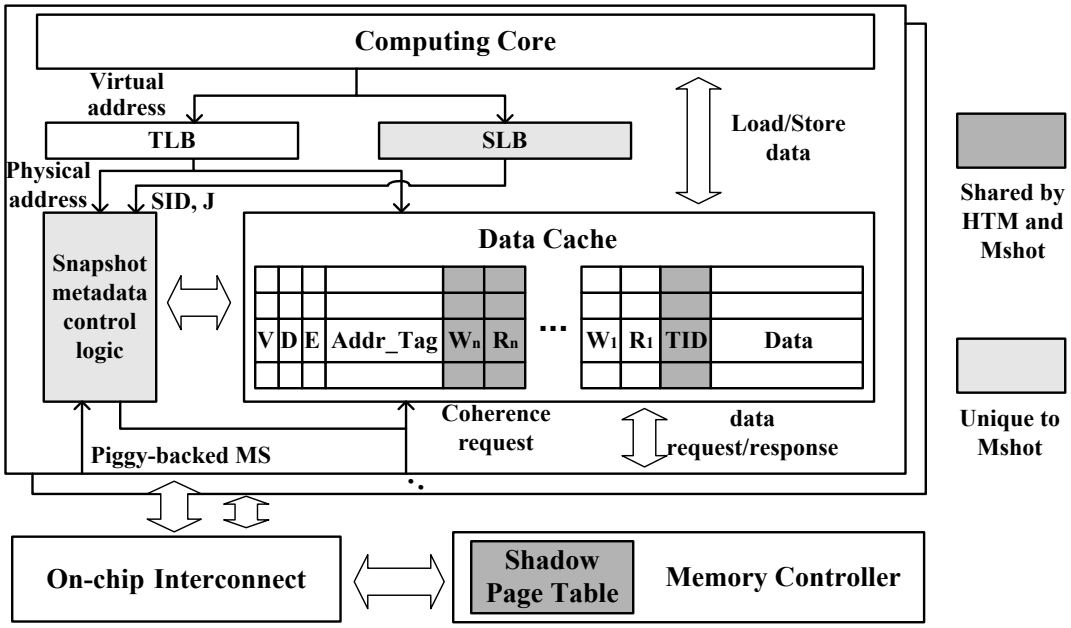

Figure 5: Resource sharing between HTM and MShot.

is compared with single-threaded call-path profiler that stops the world and analyzes the stacks [13]. The runtime overheads from garbage collection and profiling are normalized to the application execution time without running GC and profiler.

5.2 Snapshot GC

We used seven applications and a micro-benchmark for the GC tests. To observe meaningful behavior within a reasonable simulation time, a 32MB heap is used. While this is smaller than what a real environment would use, the applications still exhibit reasonable ratios of GC time over the total execution time (from 1% for Cfrac to 18% for Gzip). We also show the result from running Mpeg2 with a 128MB heap.

Figure 6 presents the runtime overheads added to the application execution time with and without snapshot. They are normalized to the execution time when running without GC by using a very large heap. Lower bars are better. In each bar, *Stop* is the time spent for stopping the system to start GC, *Mark* the time for the mark phase, *Reclaim* for the reclaim phase, and *Snapshot* the time to initiate a snapshot. *App time* is the time spent in the application itself. The MShot bar represents snapshot-based GC while the Para bar represents the parallel GC (non-concurrent).

For all applications except Nullhttpd and RBtree, snapshot GC eliminates most of the runtime overhead experienced with parallel GC. Nullhttpd and RBtree show an increase of the application execution time itself due to the increased memory contention with GC. On average, snapshot GC reduces the overhead down to 1.5%, which makes garbage collection essentially cost-free. Snapshot GC scales well with a 128MB heap for Mpeg2.

Table 5 shows the memory requirements to maintain the overflowed data for the snapshot-based GC. PTM uses additional physical pages for shadow pages even if a single cache line overflows within each page. In the table, Snapshot Data is for the actual overflowed snapshot data at cache line granularity and Shadow Page is for the number of shadow pages times the 4-KByte page size. For all applications except Nullhttpd and Mpeg2, the memory requirements for snapshot data is under 1 MByte. The worst case (Nullhttpd) is still under 7 MBytes for the 16-core CMP. Moreover, the page mappings for the shadow pages fit completely into the 2048-entry shadow page table.

5.3 Snapshot Call-path Profiler

Figure 7 shows the runtime overhead, due to call-path profiling, which is normalized to the application execution time without profiling. *Prof* is the time for running the profiler, *App* for the execution of the application itself, and *Snap* the time to take a snapshot. The profiler is triggered 50K cycles after the previous profiling has completed. Gzip, Pypy, and W3M experience the largest performance improvements with snapshot profiling ranging from 68% (Pypy) to 75% (Gzip) due to the deep call graph (average depth of 14). This result is important for programs written in object-oriented language since they typically have deeper call depth. On the contrary, Mp3d, Radix, and Cfrac have an average depth of 4 to 6. Regardless of the applications' call depth, the snapshot call-path profiler adds negligible overhead to applications (less than 3%).

6. RELATED WORK

We have discussed general alternatives to hardware-assisted memory snapshot in Section 2.1. Beyond that, there are more application-specific solutions for concurrent programs. Several software implementation schemes have been proposed recently for mostly concurrent and parallel garbage collectors [12, 20, 39]. While they are competitive to MShot in terms of low runtime overhead, MShot allows for algorithmic simplicity and easy code management as well. There are recent advances in dynamic profiling such as SuperPin and Shadow Profiling [26, 37]. However, the SuperPin paper reports overheads of 100% because of heavy-weight cloning, forking, and DBT overheads. Shadow Profiling shows lower overheads but does much simpler profiling. (e.g. count basic blocks, rather than actually scanning through heap and stack).

7. CONCLUSIONS

We propose MShot, hardware-assisted memory snapshot. MShot

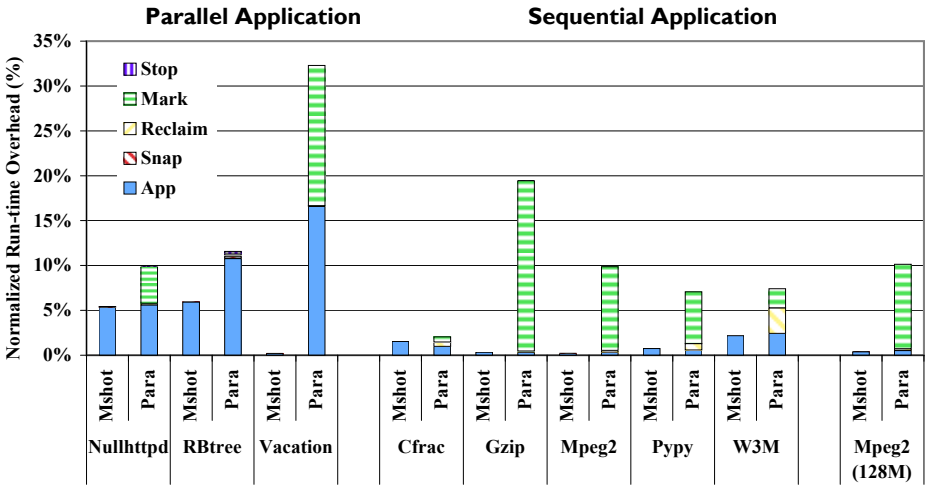

Figure 6: The graph shows the runtime overhead caused by parallel GC (Para) and snapshot GC (MShot). The overhead is normalized to the application execution time without GC. Parallel GC stops the applications, and runs with 10 cores. Snapshot GC concurrently runs with 2 cores. Sequential applications run with 1 core. Parallel GC stops the applications, and runs with 2 cores. Snapshot GC runs concurrency with 1 core.

	Nullhttpd	RBtree	Vacation-L	Cfrac	Gzip	Mpeg2	Pypy	W3M
Snapshot Data	5.25M	0.02M	0.35M	0.00M	0.59M	6.46M	0.29M	0.79M
Shadow Page	5.38M	0.28M	0.40M	0.04M	0.64M	6.52M	0.38M	0.97M

Table 5: Memory requirement to manage overflowed snapshot data.

allows programmers to read a large dataset atomically and process the snapshot image concurrently without synchronization code in multithreading environment. MShot is implemented in a cost-effective manner by using TM hardware resources. The evaluation result shows that MShot allows garbage collectors and call-path profilers to run in parallel with user applications on spare cores with negligible interference.

8. ACKNOWLEDGMENTS

We appreciate Karin Strauss and Nick George for their insightful comments on this work.

9. REFERENCES

[1] Y. Afek, H. Attiya, et al. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] J. Anderson. Composite registers. In *PODC '90: Proc. of the 9th ACM symp. on Principles of distributed computing*, 1990.

[3] A. Bilas, J. Fritts, and J. P. Singh. Real-time parallel mpeg-2 decoding in software. In *Proc. 11th Intl. Parallel Processing Symp.*, 1997.

[4] C. Blundell, J. Devietti, et al. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[5] N. S. Bowen and D. K. Pradhan. Processor and memory-based checkpoint and rollback recovery. *Computer*, 26(2):22–31, 1993.

[6] R. P. Brent. Recent progress and prospects for integer factorisation algorithms. In *COCOON '00: Proc. of the 6th Intl. Conf. on Computing and Combinatorics*, 2000.

[7] C. Cao Minh, M. Trautmann, et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *the Proc. of the 34th Intl. Symp. on Computer Architecture*. June 2007.

[8] L. Ceze, J. Tuck, et al. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture (ISCA)*, June 2006.

[9] W. Chuang, S. Narayanasamy, et al. Unbounded Page-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[10] J. Chung, H. Chafi, et al. The Common Case Transactional Behavior of Multithreaded Programs. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[11] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB '06: Proc. of the 32nd intl. conf. on Very large data bases*, 2006.

[12] D. Detlefs, C. Flood, et al. Garbage-first garbage collection. In *ISMM '04: Proc. of the 4nd intl. symp. on Memory management*, 2004.

[13] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proc. of the 19th intl. conf. on Supercomputing*, 2005.

[14] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[15] R. Garg, V. K. Garg, and Y. Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *ICS '06: Proc. of the 20th intl. conf. on Supercomputing*, 2006.

[16] http://www.gzip.org/.

[17] L. Hammond, V. Wong, et al. Transactional Memory Coherence and Consistency. In *the Proc. of the 31st Intl.*

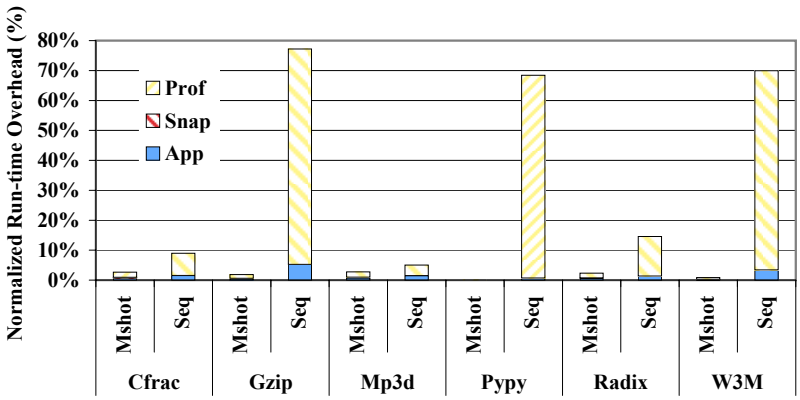

Figure 7: The runtime overhead caused by sequential call-path profiler (Seq) and snapshot call-path profiler (MShot). It is normalized to application execution time without profiling. For the sequential call path profiler, we use 1 core for both application and profiling. For the snapshot call-path profiler, we concurrently use 1 core for the application and 1 core for profiling.

Symp. on Computer Architecture (ISCA), Munich, Germany, June 2004.

[18] J. Henry C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proc. of the 1977 Symp. on Artificial Intelligence and Programming Languages*, 1977.

[19] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proc. of the 20th Intl. Symp. on Computer Architecture*, May 1993.

[20] R. L. Hudson and J. E. B. Moss. Sapphire: copying gc without stopping the world. In *JGI '01: Proc. of the 2001 joint ACM-ISCOPE conf. on Java Grande*, 2001.

[21] http://www-306.ibm.com/software/tivoli/products/storage-mgr/.

[22] P. Jayanti. An optimal multi-writer snapshot algorithm. In *STOC '05: Proc. of the 37th ACM symp. on Theory of computing*, 2005.

[23] A. McDonald, J. Chung, et al. Architectural Semantics for Practical Transactional Memory. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[24] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[25] M. J. Moravan, J. Bobba, et al. Supporting Nested Transactional Memory in LogTM. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[26] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the Intl. Symp. on Code Generation and Optimization*, 2007.

[27] http://memprofiler.com/.

[28] http://profiler.netbeans.org/.

[29] http://nullwebmail.sourceforge.net/httpd/.

[30] PowerPC Assembler Reference.

[31] http://pypy.org/.

[32] J. M. Smith and G. Q. Maguire, Jr. Effects of copy-on-write memory management on the response time of UNIX fork operations. *Computing Systems*, 1(3):255–278, 1988.

[33] L. B. Soares, O. Y. Krieger, and D. D. Silva. Metat-data snapshotting: A simple mechanism for file system consistency. In *SNAPI '03: Proc. of Intl. Workshot on Storage Network Architectur and Parallel I/O*, 2003.

[34] http://user-mode-linux.sourceforge.net/.

[35] http://w3m.sourceforge.net/.

[36] C. A. Waldspurger. Memory Resource Management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[37] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the Intl. Symp. on Code Generation and Optimization*, 2007.

[38] S. C. Woo, M. Ohara, et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita, Italy, June 1995.

[39] M. Wu and X.-F. Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *IPDPS '07: Proc. of intl. Parallel and Distributed Processing Symposium*, 2007.