Appears in the proceedings of the
16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), January 2010

HARE: Hardware Assisted Reverse Execution

Ioannis Doudalis Milos Prvulovic
Georgia Institute of Technology
{*idoud,milos*}*@cc.gatech.edu*

Abstract

Bidirectional execution is a powerful debugging technique that allows program execution to proceed both forward and in reverse. Many software-only techniques and tools have emerged that use checkpointing and replay to provide the effect of reverse execution, although with considerable performance overheads in both forward and reverse execution. Recent hardware proposals for checkpointing and execution replay minimize these performance overheads, but in a way that prevents checkpoint consolidation, a key technique for reducing memory use while retaining the ability to reverse long periods of execution.

This paper presents HARE, a hardware technique that efficiently supports both checkpointing and consolidation. Our experiments show that on average HARE incurs <3% performace overheads even when creating tens of checkpoints per second, provides reverse execution times similar to forward execution times, and reduces the total space used by checkpoints by a factor of 36 on average (this factor gets better for longer runs) relative to prior consolidation-less hardware checkpointing schemes.

1. Introduction

Debugging is an important and costly part of software development. It has been estimated [10] that debugging accounts for 60%–70% of the development effort and 80% of project overruns. It has also been estimated that in the early stages of development bugs cost an order of magnitude (50 to 200 times) [1] less to fix than in later stages. Much of the debugging time and effort is spent on back-tracking from the point where an error is manifested (e.g. the program crashes) to the point where the problem originated (e.g. an incorrect value was computed). Typical back-tracking consists of finding the variable directly responsible for error manifestation, finding where that variable was last modified, checking if that modification is a direct result of incorrect computation, and repeating this process if the modification simply propagates another incorrect value.

An example of buggy execution is shown in Figure 1(A), where a zero value in variable X causes the program to crash, e.g. with a divide-by-zero exception. Traditional back-tracking (Figure 1(B)) would involve placing a write watchpoint on X, re-running the program, inspecting the state of the program each time the watchpoint is triggered, and eventually finding the statement that placed the problematic value (zero) into X. In our example, this statement is simply "X=Y" so another back-tracking step (with a write watchpoint on Y) is needed to find where Y became zero. In this example, the buggy code is the one that sets Y to zero, so back-tracking ends there and the programmer can start figuring out how to fix that code.

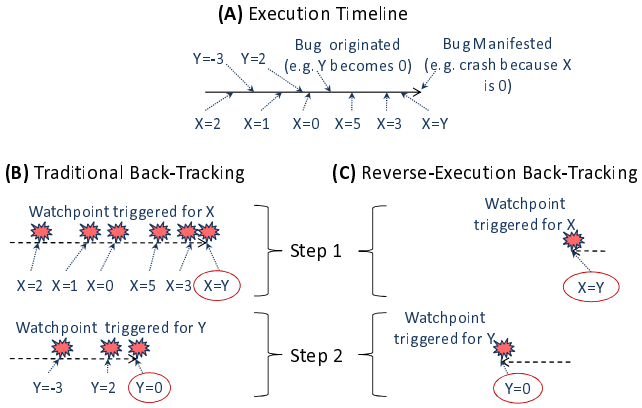

Figure 1. Debugging example with back-tracking.

This traditional approach to back-tracking is both time-consuming and labor-intensive; multiple re-executions of the program may be needed (one for each back-tracking step), often with numerous programmer interactions in each re-execution (to inspect state whenever watchpoints are triggered). Bidirectional debugging [2, 6, 9, 28] has been proposed as a powerful technique to reduce both time and effort needed for back-tracking. In addition to (traditional) forward execution, bidirectional debugging aids back-tracking by also providing backward (reverse) execution. As shown in Figure 1(C), reverse execution saves back-tracking 1) time, by only going through past execution once (not once per back-tracking step), and 2) effort, by involving the programmer only once in each back-tracking step (to inspect code that writes the watched variable).

True reverse execution is not supported in real processors and systems, so the *appearance* of reverse execution is typically implemented using checkpointing and determinis-

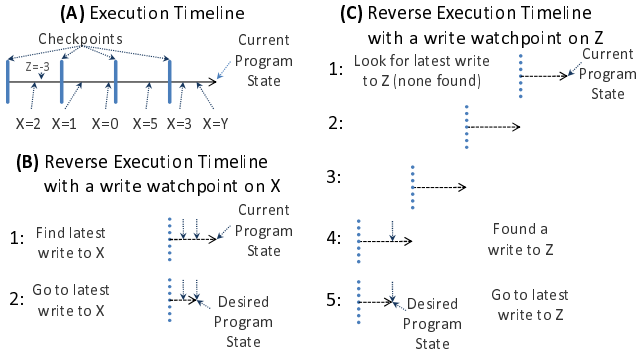

Figure 2. Reverse execution via checkpoint/replay.

tic replay, as shown in Figure 2. Checkpoints are created periodically during forward execution (Figure 2(A)). For reverse execution (Figure 2(B)), a prior checkpoint is restored, re-execution finds the latest occurrence of a watchpoint, the checkpoint is restored again, and in another re-execution we stop where the latest watchpoint was found. As shown in Figure 2(C), the most recent checkpoint interval may have no watchpoint occurrences. In this case, increasingly older checkpoint intervals are replayed until a watchpoint occurrence is found.

From the user's perspective, this checkpoint/replay implementation of reverse execution is indistinguishable from "real" reverse execution if it has similar "speed" and "reach"[1] as forward execution. For "speed", the time needed to reverse some execution should be similar to the time that was needed to forward-execute it. For "reach", we should be able to reverse-execute from any point all the way to the start of the program, just like we can forward-execute all the way to the end. Disproportionally long reverse execution times and/or inability to reverse-execute far enough into the past are likely to frustrate programmers and cause them to revert to "traditional" debugging approaches. Ability to reverse long periods of execution is also needed for analysis of security attacks that may deliberately exploit bugs with long dormancy periods.

Unfortunately, "speed" and "reach" create conflicting demands for checkpointing frequency: "speed" needs frequent checkpointing so short periods of execution can be reversed quickly, but for "reach" checkpoints should be created rarely to retain a lot of checkpoints without running out of space. To achieve both goals, some software-only tools create checkpoints frequently (for short-term "speed" of reverse execution) and then *consolidate* [2] them as they age (to conserve space over the long term). Unfortunately, software-only checkpointing approaches for reverse execution [2, 4, 5, 6, 9, 15, 19, 21, 28] have large performance overheads (when checkpointing often enough for good "speed" of reverse execution), and they cannot efficiently record memory races for deterministic replay of multi-core execution.

Hardware support has been proposed as a way to minimize performance overheads of checkpointing for error recovery [14, 17, 26] and, in combination with deterministic replay, for debugging [14, 16, 26]. Performance overheads are reduced mostly 1) by saving checkpoint data in the background and on-demand, as application execution modifies memory content, and 2) by tracking race outcomes in hardware. In recent years, memory space needed to record race outcomes has been reduced by multiple orders of magnitude [7, 12, 13, 27]. However, the total memory used for checkpoint storage is dominated by saving modified data blocks (Data Log in Figure 3), where hardware schemes could only reduce space consumption (typically by a factor of 2 to 3) using compression of individual checkpoints. In contrast, consolidation would result in orders-of-magnitude space reduction for data logs. Unfortunately, checkpoints produced by existing hardware schemes are not amenable to efficient consolidation because 1) consolidation of two checkpoints relies on finding and eliminating duplicate data blocks (those saved in both original checkpoints), which requires checkpoints to be sorted (or at least searchable) by data address, whereas 2) existing hardware schemes save data blocks on-demand, thus data blocks in each checkpoint are ordered by time of modification (or use), not by data address. As a result, these schemes must sacrifice space (by giving up on consolidation) or performance (by sorting checkpoints after they are created).

In this paper, we present HARE (Hardware-Assisted Reverse Execution), a low-cost hardware support that provides efficient checkpointing *and consolidation*. The key contribution of this work is a checkpointing mechanism that, like prior hardware mechanisms, operates in the background but, unlike prior work, produces checkpoints which are already sorted by address. We also describe a low-cost hardware checkpointing engine that can also perform checkpoint consolidation in the background, and show that this checkpointing and consolidation implementation results in minimal performance overheads, records long periods of execution using orders of magnitude less space than prior hardware schemes, and provides reverse execution times similar to forward execution times of the same instructions.

The rest of this paper reviews bidirectional debugging (Section 2), describes our technique (Section 3) and its implementation details (Section 4), and presents our quantitative evaluation (Section 5) and conclusions (Section 6).

2. A Review of Bidirectional Debugging

In addition to commands for executing the application forward (step, continue, etc.), a bidirectional debugger also provides commands for reverse execution (rstep, rcontinue,

[1] Reverse execution, its "speed", and its "reach' refer to what the user (of the debugger) perceives during bidirectional debugging.

etc.). As described in Section 1, reverse execution is typically implemented by restoring a checkpoint and then forward-executing to the desired point, using the number of executed instructions to define "position" in the execution time-line. For example, if N instructions have been executed since the last checkpoint, a `rstepi` command (undo one instruction) causes the debugger to restore the last checkpoint and re-execute N-1 instructions. In some cases, the desired point is unknown a priori. A typical example (shown in Figure 2), is when the user (of the debugger) sets a write watchpoint on some variable and then uses a `rcontinue` command (reverse-execute until watchpoint is triggered). The desired point in this case is the most recent write to the watched variable(s). The example in Figure 2(B) shows a situation where the watchpoint is triggered while replaying (phase 1) the most recent checkpoint interval. In this phase, replay continues to find if an even more recent write exists in that checkpoint interval. Once the instruction count for the last write is ascertained, another replay of the same checkpoint interval (phase 2) is used to reach that point. Alternatively, the debugger creates temporary checkpoints as it finds each write in phase 1, then phase 2 consists of simply restoring the most recent temporary checkpoint. If the watchpoint is not triggered while replaying the most recent checkpoint interval (Figure 2(C)), the debugger replays progressively older intervals until it finds one where the watchpoint is triggered.

Assuming that checkpoints can be restored as quickly as they can be created, and assuming that replay is as quick as original forward execution, the time needed for reverse execution will be equal to the time that was needed for the same execution in the forward direction, plus the time to restore and replay the rest of the target checkpoint interval (once if no temporary checkpoints are created, twice otherwise). Therefore, the "speed" of reverse and forward execution (discussed in Section 1) will be similar if the target checkpoint interval is relatively short compared to the amount of execution being reversed. For small amounts of reverse execution, e.g. reverse-executing only a few instructions, recent checkpoint intervals must be very short (so their replay appears instantaneous), thus leading to very frequent checkpointing (at least several times per second). On the other hand, long-range reverse execution achieves good "speed" even with longer checkpoint intervals.

Frequent checkpointing that can provide good "speed" for small amounts of reverse execution leads, over long periods of execution, to huge space overheads. To illustrate this, Figure 3 shows, for 8-threaded execution of PARSEC 2.0 benchmarks, the rate (in GBytes/minute) at which space is consumed by checkpoints when they are created whenever 1 billion instructions are executed (by all threads, resulting in 10-30 checkpoints per second) without using consolidation. We see that facesim, fluidanimate, freqmine, and x264 consume more than 10 GBytes per minute, so good "reach" of reverse execution would be infeasible for long-running applications (e.g. hours of execution time) with similar memory access patterns.

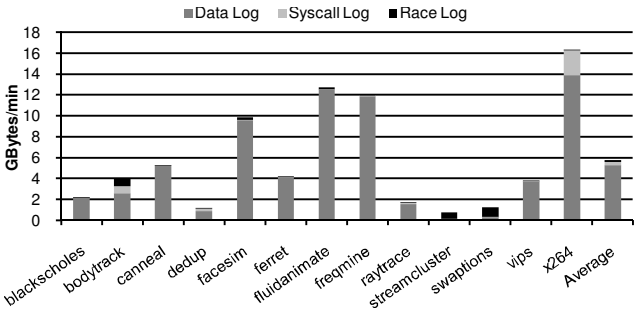

Figure 3. Space used for frequent checkpointing.

The checkpointing space requirements in Figure 3 are broken down into checkpointed data (Data Log), system event logs for deterministic replay (Syscall Log), and race logs for deterministic replay of multi-threaded execution (Race Log). We obtain Data Log and Race Log sizes by modeling the approach used in FDR [26], but without using compression (which typically can only reduce these space requirements by a factor of 2 to 3). We observe that Data Log (checkpoints themselves) is the dominant component (by far). This may seem to contradict recent research in deterministic replay, which focuses almost exclusively on reducing Race Log sizes [7, 12, 13, 27], especially knowing that PARSEC is not as I/O intensive as commercial workloads used to evaluate work that focused on recording races and *non-deterministic system events* (such as I/O) [19, 21, 26]. However, results presented for commercial workloads by Xu et al. [26] lead to similar conclusions (data logs are the dominant component).

As described previously, frequently taken checkpoints are only needed to achieve good "speed" for reversing small amounts of execution, and long-range reverse execution can achieve good "speed" using checkpoints that are further apart from each other. Some software schemes for bidirectional debugging [2] build on this insight by using *checkpoint consolidation* to dramatically reduce the total space consumed by checkpoints, while retaining good "speed" for short-range reverse execution. With consolidation, incremental checkpoints are created often, for "speed" of short-term reverse execution, but are then merged as they age into progressively coarser-grained checkpoints to reduce space requirements while still supporting good "speed" of long-term reverse execution.

A consolidated checkpoint is the union of its input checkpoints, but duplicates (data blocks present in both input checkpoints) eliminated. To efficiently find duplicates during consolidation, checkpoints must be searchable by data address or, preferably, arranged by address so a single merging pass can eliminate all duplicates. This is relatively

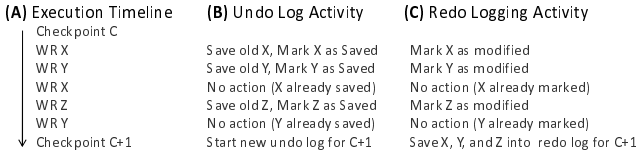

Figure 4. Checkpointing with undo and with redo logs.

easy to achieve in software schemes, which can use sophisticated data structures (e.g. search trees) to facilitate checks for duplicates. Additionally, many such schemes identify changes at page granularity [2, 6, 9, 15, 21], so there are fewer saved records to search or sort than in schemes with finer (e.g. cache block or even word) granularity.

In contrast, hardware checkpointing schemes [14, 17, 20, 26] dramatically reduce performance overheads of checkpointing by saving data on-demand (without stopping the application) and at finer (typically cache-block) granularity. Sophisticated data structures would result in prohibitive costs, so these schemes simply write out data blocks to a contiguous log in memory as each block is modified (in BugNet [14], when modified data is used for the first time). This results in checkpoint logs with many fine-grain entries that are not sorted by address. Unfortunately, this prevents efficient consolidation: these logs cannot be searched for duplicates efficiently, and sorting them is either time-consuming (if sorting in software) or requires expensive sorting hardware. It should be noted that some hardware solutions [14, 20] compress checkpoints, which typically reduces space requirements by a factor of 2 to 3. In contrast, as will be shown in Section 5.3, if efficient consolidation could be supported it would provide orders-of-magnitude reduction is total space requirements.

3. An Overview of HARE

As explained in Section 2, to support efficient consolidation we must either 1) sort checkpoints by address after they are created, or 2) create checkpoints that are already sorted by address. Sorting is time-consuming if done in software and expensive if done in hardware, so our only remaining option is to fundamentally change the checkpointing mechanism to create already-sorted checkpoints.

To efficiently create an already-sorted checkpoint, we must 1) know which blocks will be saved before we actually save any of them, and at that time 2) efficiently discover these modified blocks in order of address and save them without stopping the execution of the application. The undo-log approach used in prior hardware schemes [17, 20] is incompatible with the first requirement. Figure 4(A) shows an example execution and Figure 4(B) shows the corresponding undo log activity, which saves each old data value just before it is overwritten for the first time in this checkpoint interval. Because data is actually saved during the checkpoint interval, the next checkpoint (Checkpoint C+1) is "created" by simply marking the position in the undo log (or by starting a new log). To restore a prior checkpoint, we would simply restore to memory all values saved since that checkpoint, starting with most recent log entries. However, we only know the entire set of modified blocks at the end of the checkpoint interval - after data from all these blocks is already read out from memory and saved to the log. To produce a sorted checkpoint, we would need considerable on-chip resources to buffer these blocks (or at least records that contain pointers to saved blocks), as well as expensive hardware to sort these records by address before we can write them out as a sorted undo log. We note that BugNet [14] does not use undo logs, but its logs are sorted by time of first read and thus suffer from similar limitations when it comes to consolidation.

Due to these considerations, for our HARE scheme we forgo undo-log and loaded-value-log approaches, and use redo logs instead. Redo logging for the execution example in Figure 4(A) is shown in Figure 4(C). A redo log contains *new* values for modified data, recorded at the end of the checkpoint interval when the set of blocks that must be saved is known and can be written to the log in order of original address. However, redo logs present two main challenges that should be addressed. First, we need to track blocks that are modified during the checkpoint interval and efficiently save these modified blocks (in order of data address) when actually creating the checkpoint. Second, redo logs can directly support a *roll-forward* from older to newer checkpoints, not *roll-back* from newer to older checkpoints that is needed to provide reverse execution.

In the rest of this section we describe our modification-tracking approach, our checkpoint creation approach, how to roll back to past checkpoints created by our scheme, and how our redo logs can be efficiently consolidated, and how our logs are organized to reduce memory bandwidth needed for consolidation.

3.1. Memory Modification Tracking

Creation of checkpoint C+1 involves saving new values of all blocks that were modified between checkpoint C and checkpoint C+1. To track which blocks were modified, we use a packed bit-array with a "modified" flag (bit) for each memory block in the application's address space, and we keep, look-up, and update these per-block bits using a MemTracker-like approach [24].

When creating a checkpoint, a linear search of this bit-array can discover modified blocks in order of their (virtual) address. However, our checkpoints will be created often, so only a small fraction of blocks in the entire address space is modified, and blocks that are modified tend to be clustered (because of spatial locality of writes in the applica-

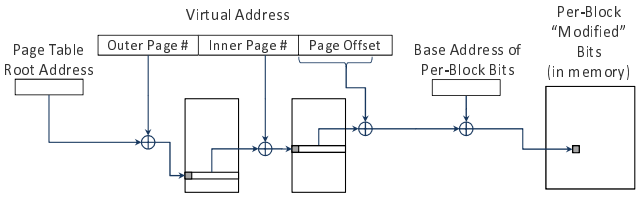

Figure 5. Modification tracking bits (shown in gray).

tion). This means that a linear search of the bit-array would be time-consuming, with most of that time spent scanning through bits that correspond to unmodified blocks.

To improve efficiency of the search for modified blocks, we also mark modifications in page tables and TLBs by using an additional dirty bit in each page table (and TLB) entry. Figure 5 shows an example with hierarchical page tables (a page table with only two levels is shown for clarity). When creating a checkpoint, modified pages can be found by identifying modified entries in outer levels, descending towards inner levels for such entries, and eventually searching only the parts of the bit-array that correspond to pages marked as modified. We clear "modified" bits in page table entries and the bit-array as we discover them and save the blocks they indicate, so when a checkpoint is created the same bits can be used to track modified blocks for the next one. Overall, for each modified page we examine at most one page table node at each level (e.g. 4 nodes for 4-level page tables) and tens of bits in the bit-array (e.g. 64 bits for 4kB page size and 64-byte blocks). Since each modified page contains at least one modified data block that must be copied to the checkpoint, the scan for modified blocks represents only a small fraction ($<$2%) of the total time and memory bandwidth used for checkpointing.

To simplify our hardware support, we use a software handler to discover modified pages and create a list of such pages. Our hardware checkpointing support then reads this list, scans corresponding parts of the bit-array, and saves discovered blocks to the checkpoint's data log. We note that, as an alternative to using additional page table bits and the bit-vector array, our modification tracking could also be implemented with a single hierarchical meta-data structure, e.g. we can use a trie structure from Mondrian Memory Protection [25] with "modified" bits at each level to quickly zero in on modified blocks. The choice of the specific mechanism to track modified blocks is largely orthogonal to the design of our checkpointing and consolidation support.

3.2. Checkpoint Creation

To create a checkpoint, we scan our modification-tracking state, find modified blocks, and save them to our redo log. In a naive implementation of this approach, the execution cannot continue until the checkpoint completed, as modification tracking for the next checkpoint can interfere with the creation of the current one, and new execution could modify data blocks again before they are saved.

We distinguish between modification marks from the previous and the current checkpoint interval by using two separate modification bits for each block and page table entry. During a particular checkpoint interval, one set of bits is used to track modifications for the next checkpoint, while the other is being scanned and cleared as blocks modified in the previous checkpoint interval are saved. The roles of the two sets of bits are reversed at each checkpoint. In the unlikely case that a checkpoint interval ends while the previous is still being saved, we stall the processor. Such stalls are not observed in any of our experiments - even with very short checkpoint intervals, checkpoint creation completes well before it is time to create the next checkpoint.

To prevent overwrites in the new checkpoint interval from destroying yet-to-be-saved values from the previous one, for each write generated by the processor we check the modification bit from the previous checkpoint interval. If that bit is still set, the block must be saved before it is overwritten. A naive solution would be to stall the processor at this point, but such stalls would occur often[2]. Instead, when a yet-to-be-saved block is about to be modified again, we save this block to another log (called *collision list*), clear its modification bit, and allow the processor's write to proceed. This leaves the main data log for the checkpoint completely sorted by address, but produces an unsorted collision list. Fortunately, our experiments indicate that the collision list is much smaller than the checkpoint list, so it can be quickly sorted in software when checkpoint creation is complete.

3.3. Restoring Checkpointed State

Bidirectional debugging restores past checkpoints in order to provide the effect of reverse execution. With undo logs, past checkpoints are restored simply by copying saved data from the undo log, in the order opposite from the one in which log entries were created. With redo logs, however, log entries contain the *new version* of each block that was modified since the previous checkpoint.

To restore some checkpoint C using redo logs, for each block modified in checkpoint intervals that follow C we must find the version of that block that existed at the point where C was created. This version is found by searching for it in checkpoint C, then in the checkpoint that immediately precedes it, etc., until it is found. This search is far more efficient than it seems because 1) most blocks are found after looking in only a few (e.g. one or two) checkpoints because writes tend to exhibit temporal locality, 2) our checkpoints are sorted by address, so binary search can be used to find if

[2]Blocks with high addresses (e.g. stack) are written often (causing a stall) and checkpointed last, so overlap between execution and checkpointing activity would be minimal

a checkpoint contains a given block, and 3) even the worst-case search (all the way to the start of execution) is fairly efficient: consolidation dramatically reduces the total number of checkpoints that are kept at any given time.

3.4. Checkpoint Consolidation

We consolidate two level-N checkpoints into a new level-(N+1) checkpoint as soon as two level-N checkpoints exist and a third is created by establishing a new checkpoint (for level-zero checkpoints) or through consolidation. This consolidation policy is similar to the one used in software-only schemes [2], and it provides two key benefits: 1) the total number of checkpoints and the total size of all checkpoints at any given time is only logarithmically proportional to the total execution since the beginning of the program, 2) checkpoints at each level of consolidation need to be consolidated only half as often as the previous level, so consolidation work grows slowly and can be bounded by saving very old (and heavily consolidated) checkpoints to disk and not consolidating them further, and 3) the "speed" of forward and of reverse execution is similar, because replay of each checkpoint interval takes about half as much time as was needed to forward-execute from the start of that interval to the "current" point in the execution.

To consolidate two (already sorted) checkpoints into a new (also sorted) one, we perform a merging pass that reads the next record from each source checkpoints and compares the addresses from which the two blocks were copied. If the addresses are the same, only the record from the later source checkpoint is copied to the consolidated checkpoint and the next record from each source checkpoints is read for the next comparison. If the addresses are different, the record with the lower address is copied to the consolidated checkpoint, while the one with the higher address is retained to be compared with the next record from the other source checkpoint. The result of this merging pass is a consolidated checkpoint that is also sorted by address.

3.5. Checkpoint Log Organization

The most straightforward checkpoint organization would be an (address-ordered) sequence of records, each containing a block of data and its original address. However, such organization would require saved data to be copied during consolidation. Instead, we separate saved data from its meta-data. Meta-data for a particular checkpoint is kept as a contiguous array of records, each with the block's original address and the pointer to the block's saved data. We also use a special "free list" meta-data array that points to free blocks in the saved-data-block space. Consolidation now leaves saved data blocks in place, and only performs a merging pass on the meta-data arrays to create a consolidated one, while records for duplicates go to the free list to be reused when new checkpoints are created.

4. Implementation Details

This section describes how HARE can be implemented and how it can be integrated with existing system and race logging approaches and with the operating system.

4.1. Caching Memory Modification Bits

Our memory modification tracking allows efficient discovery of modified blocks during checkpoint creation. However, on each store instruction the processor must look up and possibly update these structures, so they should be cached for efficiency. The "modified" bits in page table entries are equivalent to existing "dirty" bits that are used to support virtual memory, so we can extend existing TLB look-up and update mechanisms [8]. Our per-block modification bits are kept in a packed bit-array format, but can be cached in primary caches by extending the block's tag array to also keep its "modified" bits (as shown in Figure 6). This approach was called "Interleaved"caching by Venkataramani et al. [24] and was rejected as not flexible enough for their MemTracker scheme (which uses a variable number of bits per word). Since our HARE technique uses only two "modified" bits per block, it can use this simpler caching approach in L1 caches. For L2 caches, we adopt the same approach used in MemTracker - each L1 miss fetches the data and its state separately, so the L2 cache is unmodified and memory blocks that belong to the bit-array are cached normally and simply compete for L2 cache space. This is not a concern because each block of "modified" bits corresponds to many (e.g. 256 with 64-byte blocks) data blocks.

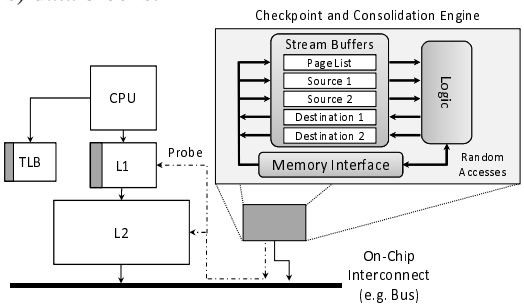

Figure 6. Hardware support for HARE, with added hardware shown as gray.

4.2. Checkpoint/Consolidation Engine

The bulk of checkpoint creation and consolidation work in our HARE scheme is implemented in a separate engine, as shown Figure 6. It has a memory interface that is used

both to 1) read data blocks from and save them to memory and 2) to provide memory access for several stream buffers. Several input and output stream buffers are used to read and write log records, and an additional stream buffer is used to read the list of modified pages (see Section 3.1).

When creating a checkpoint, the engine reads the addresses of modified pages, fetches the corresponding per-block "modified"bits from the bit-array, and finds which blocks are marked as "modified". For each such block, it gets the next free-list record, fetches the block from memory and saves it to the space indicated by the free-list, and adds a new record (with the block's original address and the address it was written to) to the checkpoint's log. To get up-to-date values, e.g. when reading modified data blocks that may still be in the writer's cache, the HARE engine must participate in coherence, generating read requests to obtain data blocks (from either memory or caches) and generating invalidations for memory blocks that are written (logs and clearing of "modified" bits).

When consolidating two checkpoints (as described in Section 3.4), the two input stream buffers are used to read records from their logs, and output stream buffers are used to write out the consolidated checkpoint log and to add entries to the free list. However, after a checkpoint is established by our scheme, it consists of two sorted logs (already-sorted data log and software-sorted sorted collision list). To consolidate two such "bifurcated" checkpoints, we can either 1) "consolidate" each checkpoint's data log with its own collision list, then consolidate the two actual checkpoints in another merging pass, or 2) extend our HARE engine with two more input stream buffers and additional logic to handle a four-way merge. In our experiments we use the second approach (four-way merge) to simplify our software handlers and to achieve more efficient consolidation.

4.3. System and Race Logging

Our HARE mechanism focuses on creating data checkpoints and efficiently consolidating them. It can be used together with existing hardware data race recording mechanisms [7, 12, 13, 26, 27] to allow accurate deterministic forward and backward execution for debugging in multi-core machines. Similarly, non-deterministic system events must be recorded so they can be accurately replayed. The type of event and its timing can be captured by the system at the point when control is transferred (via interrupt, I/O system call, etc.) from the application to the system code. However, changes made by the system in the application's address space must also be captured, and it would be impractical to instrument the operating system to track such changes. Previous hardware implementations [14] capture these modifications by terminating the current checkpoint interval and establishing a checkpoint just before the system event is executed, processing the event normally, then establishing a new checkpoint after the event completes.

In our HARE mechanism, we already have two sets of "modified" bits, which allows us to capture system-event modifications without terminating the current checkpoint in the application. When the system event (e.g. system call) occurs while one set of bits is clear (the previous checkpoint is complete), we use that "free" set of bits to track memory modifications while in system code. When returning to the application, we can "checkpoint" only those changes to the system event log (which also resets these "spare" modification-tracking bits), then continue tracking changes in the application's current checkpoint interval using the original set of "modified" bits.

If a system event occurs while both sets of "modified" bits are in use (the application's previous checkpoint is still being created), we simply stall the application until that checkpoint is complete, at which time one set of "modified" bits is free. Note that this stall would have to happen even if we used the approach of ending a checkpoint interval before the system event - we would need a third set of "modified" bits to continue executing while *two* prior checkpoints are still being created.

Finally, we note that, because race and system logs cannot be consolidated, their size grows linearly as execution proceeds. Data logs, which without consolidation would also grow linearly and represent most of the overall memory consumption, now grow only logarithmically. Over long periods of time (e.g. minutes or hours) consolidation-reduced data logs eventually become smaller than the ever-growing race and system logs, thus putting priority again on improvements in race and system log recording. In effect, consolidation is *necessary* to keep data log sizes at bay and allow race and system log optimizations to have an impact.

4.4. OS Interaction

Our HARE approach uses Operating System (OS) software handlers to reduce hardware complexity and ensure correct operation. We rely on the OS to determine (typically via timer interrupts) when to create a new checkpoint. At that time, the OS walks the page tables and builds a list (actually, a contiguous array) of modified pages. It then modifies control registers to switch to the other set of modification tracking bits, configures the checkpoint/consolidation engine to start checkpoint creation, and allows the application to continue. If the engine is still creating a previous checkpoint, for that application, the application is suspended until that checkpoint is complete. If the engine is busy doing consolidation, the OS saves the internal state of the engine and starts it on checkpoint creation (consolidation can be delayed without stalling the application). The engine generates an interrupt when its current task is com-

plete, at which point the OS checks if another task (consolidation or checkpointing) should be initiated.

Another OS-related issue arises when swapping out a page that is marked as modified in the current checkpoint - HARE's engine will not be able to read the block's data. If the OS keeps a sufficiently large free-page pool, we can delay page eviction until the end of the checkpoint interval (with our frequent checkpointing, this is only a minor delay). For urgent page evictions, modified blocks can simply be saved to the conflict list prior to eviction.

5. Evaluation

We quantitatively evaluate our HARE support in terms of estimated hardware cost, performance overhead in forward execution, memory requirements for long-term checkpointing, and reverse execution time.

5.1. Hardware Cost

In this evaluation, we model a multi-core system, with Core2-like parameters: 4-issue, out-of-order cores running at 2.93GHz. Each core has a 32KByte, 8-way set-associative, dual-ported L1 data cache, and all cores share a 4MByte, 16-way set-associative, single-ported L2 cache. Block size is 64 bytes in all caches, and the processor-memory bus is 64 bits wide and operates at 800MHz. For our HARE mechanism, we model an on-chip checkpoint/consolidation engine that participates in coherence and has (fully snooped) 16-entry read and write queues. As described in Section 4.4, the engine only works on one checkpointing or consolidation task at a time, but checkpointing tasks can preempt consolidations. Using CACTI 5.3 [23], we estimated that the overall internal state of HARE's checkpoint/consolidation engine (read and write queues, stream buffers, internal latches, etc.) uses less than 5% of the area occupied by one L1 cache, while caching of our "modified" bits adds a $<$1% overhead to each L1 cache. Finally, the bit-array for out modification tracking is stored in memory and uses only 0.4% of the memory used by the application being debugged. Overall, hardware support for our HARE mechanism amounts to about 4KB of fast on-chip SRAM and represents a negligible fraction of the overall chip area. Its cost is considerably lower than 48KB reported for BugNet [14] or 1416KB for FDR [20], so we expect HARE to add little cost when it is combined with existing race logging mechanisms.

5.2. Performance Overhead

For our performance evaluation, we used SESC [18], an open source execution driven simulator, to model a 4-core system with parameters described in Section 5.1. For each simulated application, we skip 5% of the execution and then simulate for 16 checkpoint intervals. We use 23 of the 29 SPEC 2006 [22] benchmarks; perlbench, lbm, milc, sphinx3, tonto, and zeusmp are omitted because of incompatibilities (mostly from x86-specific code) with our simulator infrastructure. We also evaluate HARE with multi-threaded workloads, using a subset of Splash-2 and PARSEC 2.0 suites, with inputs shown in Tables 1. The remaining Splash-2 benchmarks are omitted because no input sets are available that result in sufficiently long execution (16 checkpoints), whereas the omitted PARSEC benchmarks have x86-specific code incompatible with our simulation infrastructure.

Benchmark	Input
Splash-2	
Barnes	64K
FMM	64K
LU	4096x4096
Radiosity	-room
Water-sp	4096
Water-n2	4096
PARSEC 2.0	
Blackscholes	64k options
Bodytrack	4 cameras, 2 frames, 2000 particles, 5 layers
Facesim	372,126 tetrahedra, 1 frame
Fluidanimate	300000 particles, 5 frames
Swaptions	64 swaptions, 20000 simulations

Table 1. Multi-threaded benchmarks and their inputs.

Figure 7 presents the performance overhead of HARE when checkpoints are established every 100 million instructions. For each benchmark suite, we only show the average (over all applications we simulated) and the three applications that had the worst overheads for HARE. We also show the performance overhead that would result from using prior approaches with checkpoint consolidation. *Copy On Write* models a software-only page-granularity undo logging scheme, where all pages are marked as read-only when a checkpoint is created.The first write to each page causes a protection fault, and the fault handler saves the old version of the page, changes its permission to allow writes, and resumes the application. In this approach, most of the overhead comes from delaying application execution while copying pages. The overheads of sorting lists of saved pages for consolidation is present, but it is dwarfed by page copying overheads. *Undo Log* models a block-granularity hardware undo logging scheme such as FDR [20] or ReVive [17], using software handlers to sort lists of saved blocks prior to consolidation. In this case, the overhead is dominated by sorting of these lists [3].

[3] Sorting overhead is much higher in Undo Log than in Copy On Write because a modified page often has numerous modified blocks, so per-block logs tend to have many times as many entries as per-page ones do.

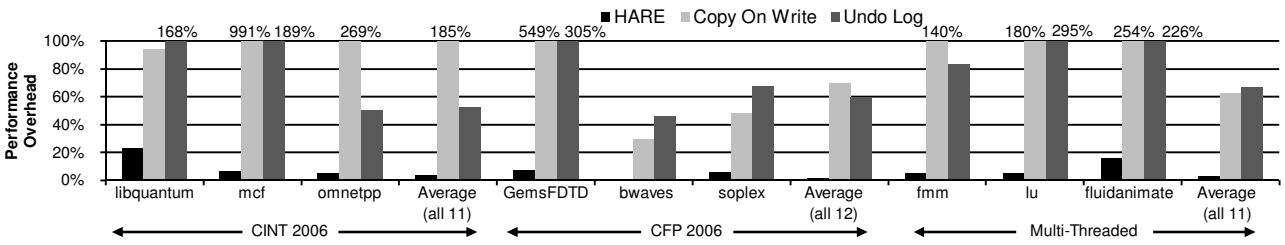

Figure 7. Performance overhead when checkpointing every 100 million instructions.

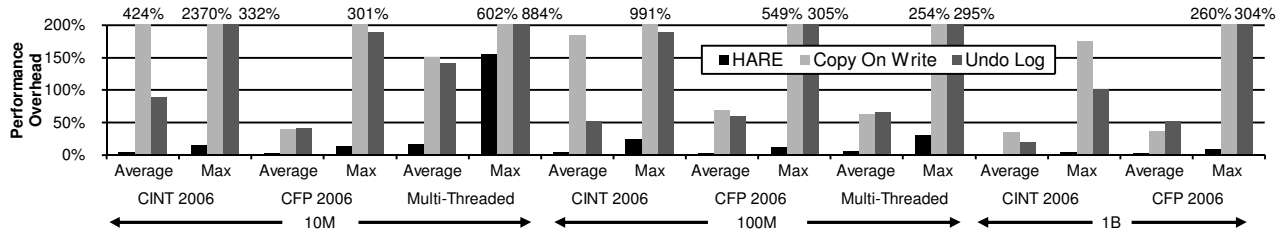

Figure 8. Performance overheads when consolidation is used.

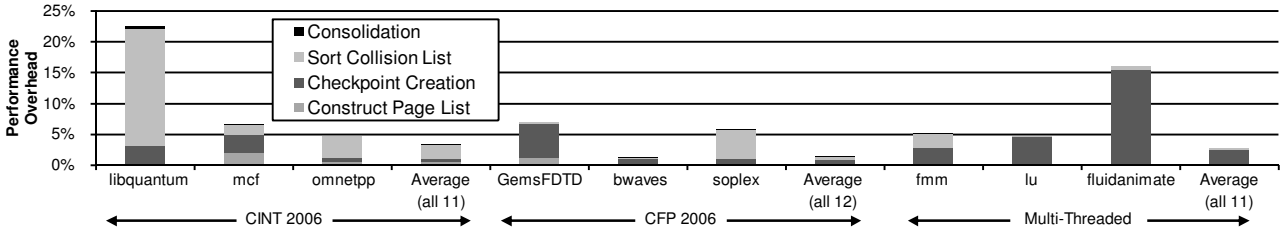

Figure 9. Breakdown of performance overheads for our HARE mechanism.

Overall, HARE incurs a maximum overhead of 23% across all applications, with averages under 3% for SPECint, under 1% for SPECfp, and under 3% for multi-threaded applications. Software-only checkpointing and consolidation results in averages that can exceed 100% (for SPECint and SPECfp), with overheads in individual applications as high as 991% (in mcf). Hardware undo log checkpointing with software sorting for consolidation still results in high overheads both on average (>50% overhead in each of the three sets of benchmarks) and on individual applications (>300% overhead in GemsFDTD).

To examine how checkpointing frequency affects performance, Figure 8 shows results from additional experiments with different checkpointing intervals. We observe that overheads in HARE do increase when checkpointing is very frequent (every 10M instructions), such that some applications suffer >100% overheads. On the other hand, when checkpointing intervals are increased to 1 billion[4] instructions (only a few checkpoints per second), both the maximum and the average overhead in HARE is reduced significantly. Surprisingly, the overhead of *Undo Log* is still significant (average >50% and maximum >300% in SPECfp). This is because, for relatively frequent checkpointing, longer checkpointing intervals can result in proportionally larger logs to sort for each checkpoint, and sorting time grows super-linearly with log size.

To gain insight into the sources of HARE's performance overhead, Figure 9 breaks this overhead down into four components (bottom to top): *Construct Page List* is the time needed for our software handlers to scan the page table and find modified pages for each checkpoint; Checkpoint Creation is the increase in execution time due to using HARE's engine to create a checkpoint (most of the overhead is due to contention for cache and memory bandwidth between the processor and the engine); *Sorting Collision List* is the overhead of sorting (in software) the collision list after a checkpoint is created; *Consolidation of checkpoints* is the increase in execution time due to using HARE's engine to consolidate checkpoints (again, most of this overhead comes from cache and memory bandwidth contention).

Dominant sources of performance overhead in HARE are sorting of the collision list (e.g. in libquantum) and memory and bandwidth contention from checkpointing (e.g. in fluidanimate). Consolidation causes minimal over-

[4]Note that we do not show results for multi-threaded applications with 1 billion instructions per checkpoint – the input sets we simulated in these benchmarks (Table 1) do not execute long enough to create sixteen checkpoints with this checkpointing interval.

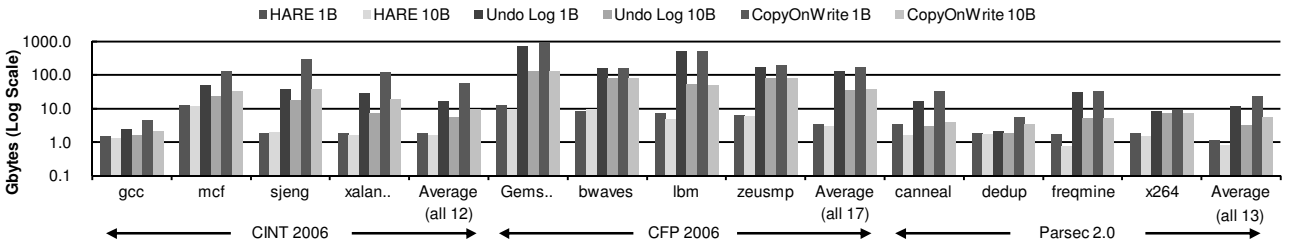

Figure 10. Total checkpoint size (in gigabytes) for checkpointing every 1 billion and every 10 billion instructions

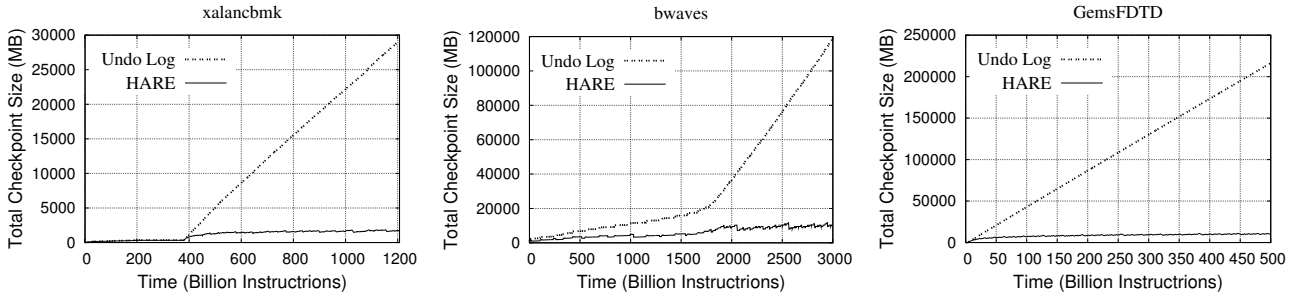

Figure 11. Total checkpoint space over time.

heads, because it operates only on meta-data (saved block's data is not moved) and uses much less bandwidth than checkpoint creation.

Overall, we find that the overheads of HARE are significantly lower than when consolidation is added to previous approaches. HARE's overheads are also low enough to permit debugging of long-running applications with realistic (large) data sets.

5.3. Checkpoint Space Overheads

To estimate long-term memory requirements, we used PIN [11] to model[5] HARE's functionality. We then used all SPEC 2006[22] benchmarks, with *ref* input sets, and all PARSEC 2.0 [3] benchmarks, with *native* inputs and 8 threads. Each application is simulated to completion, using regular checkpoint consolidation in HARE and no checkpoint consolidation in *Copy on Write* and *Undo Log*. Note that the *Copy on Write* and *Undo Log* we show here are not the same as in our performance overhead experiments – without consolidation, overheads in these schemes would be significantly lower, and with consolidation their space overheads would be similar to HARE's. We use these different kinds of experiments in different parts of the evaluation to show that HARE achieves both efficient space use and low forward-execution overheads, while prior *Undo Log* hardware schemes can only achieve one or the other – either 1) low performance but high space overheads if consolidation is not used, or 2) low space but high performance overheads if consolidation is used.

[5]Detailed simulation is infeasible for this, as it goes through only a few checkpoint periods per hour of simulation time.

The result of this experiment is shown in Figure 10. We observe that total space use for checkpoints in HARE is an order of magnitude lower than in prior hardware schemes (note the logarithmic scale in these charts). To reduce space requirements, other schemes can try reducing checkpointing frequency – as shown in these charts, reducing the frequency from once per 1 billion instructions (several times per second) to once per 10 billion instructions (once every several seconds, with frustrating delays for short reverse execution) does significantly reduce total space use in *Undo Log* schemes (FDR or ReVive). Even then, however, the space use in these schemes can reach hundreds of gigabytes to capture what amounts to tens of minutes of execution.

In contrast, HARE's consolidation effectively changes the checkpoint frequency for older checkpoints, allowing it to use a limited amount of space regardless of checkpointing frequency. In most applications, HARE with checkpointing every 1 billion instructions or every 10 billion instructions consumes similar amounts of space. More importantly, HARE with checkpointing every 1 billion instructions often uses an order of magnitude less space than when consolidation-less *Undo Log* or *BugNet* checkpoint every 10 billion instructions.

It should be noted that HARE's advantage over consolidation-less schemes grows as execution proceeds: Figure 11 shows total space use over time in HARE and in *Undo Log* for several applications. The trend is toward linear growth in *Undo Log* and logarithmic growth in HARE. In each chart, we only show data for a limited number of instructions because in entire-execution charts HARE's line becomes hard to distinguish from the X-axis.

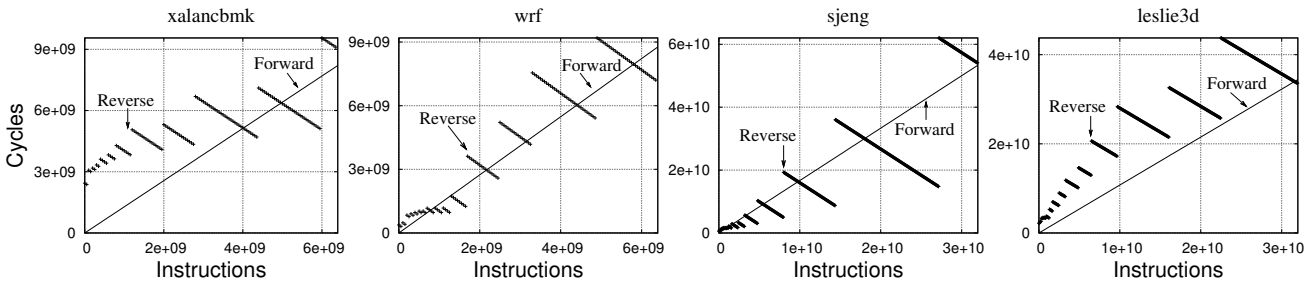

Figure 12. Estimates of reverse execution time and forward execution time with HARE.

5.4. Reverse Execution Time

To evaluate whether HARE achieves the goal of supporting the programmer's intuitive notion that reverse and forward execution should have similar "speed", we perform additional experiments where we estimate the latency of varying amounts of reverse execution (from back-stepping one instruction to undoing the entire execution), all starting from the same state (the very end of the application's execution). It would take years to get these results through cycle-accurate simulation, so for this evaluation we estimate the time of reverse execution as follows: we use profiling to determine the time required to search an average-sized checkpoint for a particular block, and how much time is needed to restore a block from checkpoint space to application's memory; we use our experiments to collect data on how many checkpoints need to be searched on average to find a desired block, and how many blocks would have to be restored in each reverse execution; finally, with these per-event times and event counts we compute estimated reverse execution times. These results are shown in Figure 12, along with forward-execution times. We only show the four application with the worst "speed" of reverse execution relative to forward execution.

As expected, the latency of forward execution grows in linear proportion to the number of instructions executed. For backward execution, however, this latency is "choppy". This effect is caused by how the checkpoint-and-replay implementation of reverse execution works: it restores a checkpoint that precedes a target position, then re-executes until that target position is reached. For target positions within the same checkpoint interval (the same checkpoint is restored), replay time *grows* as we reverse-execute *fewer* instructions - reversing to the beginning of the checkpoint interval results in negligible replay time, while reversing to the end of the checkpoint interval requires replay of the entire checkpoint interval. As a result, reverse execution time shown in Figure 12 "jumps" whenever the increase in the number of instructions results in changing the checkpoint that needs to be restored, but then smoothly improves as fewer instructions in that checkpoint interval are replayed.

The overall trend shown in Figure 12 still matches what the user (of the debugger) expects from long-term reverse execution: its "speed" is similar (well within a factor of 2) to forward execution. For short-term reverse execution, the intuitive expectation is for it to be "instantaneous" - forward execution of the same number of instructions takes negligible time. The largest difference in "speed" between forward and reverse execution occurs when executing (or reversing) only one instruction (leftmost point in each chart): a forward single-step of one instruction takes only nanoseconds, whereas the worst latency we observe for reverse single-step is about 1 second (in xalancbmk, see the leftmost chart in Figure 12). Although not truly below the human perception threshold, this is still a highly interactive response that is unlikely to frustrate programmer, and it can be improved by checkpointing more often (at the cost of some additional performance loss in forward execution).

6. Conclusions

Bidirectional debugging is a powerful debugging technique that allows program execution to proceed both forward and in reverse. Many software-only techniques and tools have emerged that use checkpointing and replay to provide the effect of reverse execution, although with considerable performance overheads in both forward and reverse execution. Recent hardware proposals for checkpointing and execution replay minimize these performance overheads, but in a way that prevents checkpoint consolidation, a key technique for reducing memory use while retaining the ability to reverse long periods of execution.

This paper presents HARE, a low-cost hardware technique that efficiently supports both checkpointing and consolidation. Our experiments show that HARE incurs small ($<3\%$) performace overheads even when checkpointing several times per second, provides reverse execution times similar to forward execution times, and reduces the total space used by checkpoints by a factor of 36 on average (this factor gets better for longer runs) relative to prior consolidation-less hardware schemes.

7. Acknowledgements

This work was supported, in part, by National Science Foundation (NSF) grants 0447783, 0541080, 0903470, and 0916464, and by the Semiconductor Research Corporation (SRC) contract 2009-HJ-1977. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and/or SRC.

References

[1] B. Boehm and P. Papaccio. Understanding and controlling software costs. *Soft. Eng., IEEE Trans. on*, 14(10):1462–1477, 1988.

[2] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *ACM SIGPLAN 2000 Conf. on Prog. Lang. Design and Impl.*, pages 299–310, 2000.

[3] Christian Bienia and Sanjeev Kumar and Jaswinder Pal Singh and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th Symp. on Operating Sys. Design and Impl.*, pages 211–224, 2002.

[5] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *4th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, pages 121–130, 2008.

[6] S. I. Feldman and C. B. Brown. IGOR: a system for program debugging via reversible execution. *SIGPLAN Not.*, 24:112–123, 1989.

[7] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *35th Intl. Symp. on Comp. Arch.*, pages 265–276, 2008.

[8] Intel. Intel 64 and IA-32 Architectures Application Note TLBs, Paging-Structure Caches, and Their Invalidation. *http://www.intel.com/design/processor/applnots/317080.pdf*, 2008.

[9] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines . In *USENIX 2005 Tech. Conf., General Track*, page 1–15, 2005.

[10] A. Kolawa. The Evolution of Software Debugging. In *http://www.parasoft.com/jsp/products/article.jsp?articleId=490*, 1996.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN 2005 Conf. on Prog. Lang. Design and Impl.*, pages 190–200, 2005.

[12] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. *Proc. of the 35th Intl. Symp. on Comp. Arch.*, pages 289–300, 2008.

[13] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *12th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 229–240, 2006.

[14] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32nd Intl. Symp. on Comp. Arch.*, 2005.

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *USENIX 1995 Tech. Conf. Proc. on USENIX 1995 Tech. Conf. Proc.*, pages 18–18, 1995.

[16] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *30th Intl. Symp. on Comp. Arch.*, pages 110 – 121, 2003.

[17] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *29th Intl. Symp. on Comp. Arch.*, pages 111–122, 2002.

[18] J. Renau et al. SESC. *http://sesc.sourceforge.net*, 2006.

[19] Y. Saito. Jockey: A User-space Library for Record-replay Debugging. In *6th Intl. Symp. on Automated Analysis-driven Debugging*, pages 69–76, 2005.

[20] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *29th Intl. Symp. on Comp. Arch.*, pages 123–134, 2002.

[21] S. M. Srinivasan, S. Kandula, and C. R. Andrews. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Tech. Conf., General Track*, page 29–44, 2004.

[22] Standard Performance Evaluation Corporation. SPEC Benchmarks. *http://www.spec.org*, 2006.

[23] S. Thoziyoor et al. Cacti 5.3. *http://quid.hpl.hp.com:9081/cacti/*, 2008.

[24] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *13st Intl. Conf. on High-Perf. Comp. Arch.*, pages 273–284, 2007.

[25] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *10th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 304–316, 2002.

[26] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *30th Intl. Symp. on Comp. Arch.*, pages 122–135, 2003.

[27] M. Xu, M. D. Hill, and R. Bodik. A Regulated Transitive Reduction (RTR) For Longer Memory Race Recording. In *12th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, volume 34, pages 49–60, 2006.

[28] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *34th Intl. Symp. on Comp. Arch.*, 2007.