Appears in the Workshop on
Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE), March 2011.

HARE++: Hardware Assisted Reverse Execution Revisited

Ioannis Doudalis
Georgia Institute of Technology
idoud@cc.gatech.edu

Milos Prvulovic
Georgia Institue of Technology
milos@cc.gatech.edu

Abstract

Bidirectional debugging is a promising and powerful debugging technique that can help programmers backtrack and find the causes of program errors faster. A key component of bidirectional debugging is checkpointing, which allows the debugger to restore the program to any previously encountered state, but can incur significant performance and memory overheads that can render bidirectional debugging unusable. Multiple software and hardware checkpointing techniques have been proposed. Software provides better memory management through checkpoint consolidation, while hardware can create checkpoints at low performance cost which unfortunately are not consolidation-friendly. HARE was the first hardware technique to create consolidatable checkpoints in order to reduce the overall performance and memory costs.

In this paper we are proposing HARE++, a redesign of the original HARE that addresses the shortcomings of the original technique. HARE++ reduces the *reverse* execution latency by 3.5-4 times on average compared to HARE and provides the ability to construct both undo and redo-log checkpoints allowing the seamless movement over the execution time of the program. At the same time HARE++ maintains the performance advantages of HARE, incurring $< 2\%$ performance overhead on average across all benchmarks, and allows the consolidation of checkpoints with memory requirements similar to HARE.

Keywords Checkpointing, debugging, hardware-accelerators.

1. Introduction

Debugging is a time consuming, yet important, process during software development. It has been estimated [11] that developers consume 60%–70% of the development time in debugging and that 80% of project overruns are caused by software bugs. It has also been estimated [1, 2] that fixing a bug in the early stages of development has a cost of 50 to 200 times less than identifying and fixing it at a later development stage, and that 80% of program code rework is being caused by 20% of the bugs. *Bidirectional debugging* [3, 9, 18] is one of the promising debugging techniques for accelerating the debugging processes and assisting the programmer find the causes of bugs. Bidirectional debugging provides the ability to execute the program both in *forward* and *reverse* program order, allowing the user to trace back in time the sources of the different data values faster and more efficiently and isolate the cause of a bug quicker.

Today's systems cannot execute a program in reverse order, and the illusion of "reverse" execution is provided by a combination of checkpointing and deterministic replaying as shown in Figure 1. For example, if the user wants to execute a "reverse step" command to back-track a single instruction in the past, the system is going to restore the program state to the closest checkpoint (Chpt 4) and replay *n-1* instructions. For the case of a watchpoint, where the user wants to perform a "reverse continue" and find the last time *X* is being modified, the system is going to search all previous checkpoint intervals (by re-executing the program) until it finds the last time *X* is modified (X=2), and replay the final interval a last time until it reaches the point of interest.

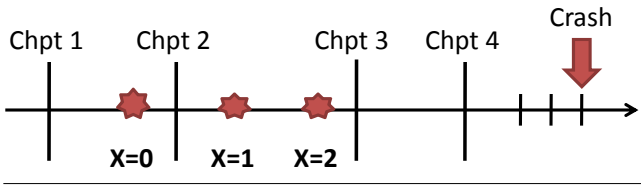

Figure 1. Reverse execution examples.

A critical mechanism necessary for reverse execution is checkpointing, whose performance and memory overheads affect the applicability and usability of reverse execution. Ideally, checkpointing for *reverse* execution should have the following two characteristics: First, it should be frequent, e.g. every second, in order to reduce the latency of reverse execution. Events such as "reverse step" should appear instantaneous to the user and the debugging experience should be interactive. Second, the memory requirements should be tolerable during debugging and allow the user to ideally debug long-running applications (hours of execution) for which bidirectional debugging could prove most useful. There are both software [3, 7, 8, 9, 14, 17, 18, 20] and hardware [15, 23] supported implementations of checkpointing for bidirectional debugging. The shortcoming of software solutions is the high performance overhead when creating checkpoints at high frequencies, an approach that is typically avoided, resulting in high reverse execution latencies. Software techniques, though, have managed to overcome the memory overhead problem efficiently and reduce the memory requirements of checkpointing through checkpoint consolidation [3]. Checkpoint consolidation creates the union of addresses of two checkpoints and removes any duplicate entries. Hardware techniques [15, 19, 23] can deliver frequent checkpointing, by efficiently copying memory in parallel with the program execution, and assist in the recording of thread interactions for replaying multi-threaded applications. Unfortunately, hardware techniques do not create checkpoints which are amenable to consolidation and as a result they suffer from high memory requirements for the case of long running applications.

To overcome both the performance and memory overheads of frequent checkpointing Doudalis et. al proposed HARE [6]. HARE is a hardware assisted checkpointing technique that creates checkpoints in a consolidation friendly format and uses a hardware engine to allow frequent checkpoint creation and efficiently overlap the program execution with the checkpoint creation process. HARE's ability to consolidate checkpoints came at the cost of increased *reverse* execution latency compared to other hardware techniques. HARE selected to create redo-log checkpoints, which can be efficiently consolidated, while prior hardware techniques [13, 15, 23] had selected undo-log checkpoints, which are not consolidation friendly but allow the system to restore the program quickly

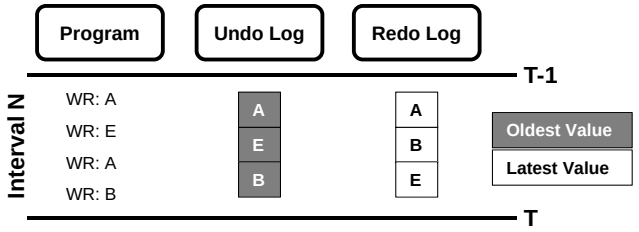

Figure 2. Examples of undo and redo-log checkpoints.

to a past point in time. To implement *reverse* execution HARE has to convert the redo-log checkpoints to undo-log, a process that results in increased latency.

In this paper we are proposing HARE++, a redesign of the original HARE [6] technique and we make the following contributions:

- HARE++ constructs undo-log checkpoints, using meta-data that allow checkpoint consolidation by hardware, and efficiently reduces the latency of *reverse* execution.
- HARE++ exploits the synergies between undo and redo-log checkpoints to also create redo-log checkpoints, while reducing the associated performance cost and requiring no additional memory modification tracking mechanism.
- HARE++ allows the creation of checkpoints at high frequencies, e.g. every 0.1sec, at low performance cost, less $< 2\%$ on average across all benchmarks, and outperforms hardware solutions that combine HARE with other undo-log checkpointing hardware techniques.

The rest of the paper is organized as follows: In Section 2 we describe existing checkpointing techniques, then we give an overview of HARE++ (Section 3) and describe the implementation details (Section 4). In Section 5 we discuss our performance evaluation results and finally we present our future work and conclusions (Section 6).

2. Review of Checkpointing Techniques

Memory checkpoints are typically used for reliability [15, 19] and debugging purposes [6, 18, 23] and can be constructed using 3 methods. The first, and most expensive, method for creating a memory checkpoint is to periodically stop the application's execution and make a full copy of the address-space of the application, a process that clearly has high performance and memory overheads and cannot be repeated often. The second checkpointing method that can efficiently reduce the overheads and increase the checkpointing frequency, is to create incremental checkpoints, also called redo-log checkpoints (Figure 2). To create a redo-log checkpoint, we keep track of the modified memory locations during interval *N* and at the end of the interval we copy all modified memory locations and store their latest values. Using redo-log checkpoints, we can restore the program from a past state *T-1* to a future one *T*, which moves the program state forward in time. Finally, the third checkpointing method is undo-log checkpointing, which records the old value of a memory location when it is modified for the first time in a given checkpoint interval. Undo-log checkpoints allow to restore the program from the current point in time (*T*) to a past one (*T-1*).

The majority of prior work in bidirectional debugging and reliability typically creates undo-log checkpoints for quick recovery to a past state. Software techniques have implemented checkpointing at the level of application [4], run-time library [14], operating system [20], or virtual machine [18]. Software typically keeps track of memory modifications at the level of a page, and leverages *copy-on-write* in order to identify the memory locations that get modified for the first time and checkpoint them. The coarse memory tracking granularity that software techniques are forced to use proves especially problematic when trying to create checkpoints at a high frequency, resulting typically at high performance overheads. As demonstrated by the results of HARE [6], at high checkpointing frequencies the application's memory is often sparsely modified, resulting in unnecessary blocks being checkpointed, which in turn increase the performance overhead. Hardware techniques [15, 19] can efficiently create checkpoints at high frequencies by keeping track of memory modifications at a finer memory tracking granularity, e.g. block, and they can overlap the checkpoint creation process with the application execution.

Software techniques have proposed checkpoint consolidation [3] as an efficient method to reduce the memory requirements of checkpointing while maintaining the ability to *reverse*-execute at any point in time. Consolidation creates the union of two checkpoints and removes any duplicate entries. When undo-log checkpoints are being created, their entries are inserted in the order of addresses getting modified by the program, and consolidation would require to search for every entry of checkpoint *A*, if there is the same address in checkpoint *B*. Sorting the checkpoints by address and then doing a merge of the two sorted lists would reduce the overall consolidation cost. For the case of software techniques, which use page granularity and have a small number of checkpoint meta-data entries, this process has low cost. Hardware techniques, however, track memory modifications at a memory block granularity and have larger checkpoint meta-data, so they experience high performance overheads [6] if they follow the same approach.

To overcome the high consolidation cost that the use of undo-log checkpoints would entail, HARE [6] selected to use redo-log checkpoints, which can be constructed to be sorted by address (at checkpoint creation time, all addresses we are going to checkpoint are known). Using a hardware engine, HARE managed to deliver low performance overheads at high checkpointing frequencies and enabled efficient checkpoint consolidation. As indicated before, redo-log checkpoints restore the program from a past to a future state, while reverse execution requires undo-log checkpoints which can quickly restore to a past program state. For this reason, HARE has to convert its redo-log checkpoints to undo-logs [6], a process that increases the latency of *reverse* execution. Other design disadvantages of HARE are that it requires frequent intervention of software during checkpoint creation, e.g. for generating the list of pages to search for modified memory blocks, or to sort the collision list.

3. Overview of HARE++

In Section 2 we described the shortcomings of the original HARE [6] technique. The goals of our current work is to improve HARE: 1) Reduce the reverse execution latency, 2) propose a hardware technique that is self-contained and requires fewer modifications to hardware structures, e.g caches, and 3) does not require frequent software intervention. To reduce the reverse execution latency we want to extend HARE to directly construct undo-log checkpoints, but still maintain the original memory benefits of checkpoint consolidation. At the same time, we want to extend HARE to provide similar functionality as software techniques [18]: construct both undo and redo-log checkpoints, but with limited additional performance and memory costs.

3.1 Undo-Log Construction

The primary reason that HARE [6] chose the creation of redo-log checkpoints over undo-logs was because redo-log checkpoints can be constructed to be sorted by address (Figure 2), while the entries of undo-log checkpoints are inserted by the order of modification, resulting in unsorted checkpoints. Consolidating undo-log check-

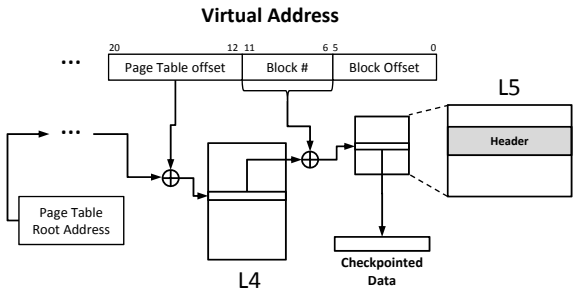

Figure 3. Trie data-structure used by HARE++ for representing the undo-log checkpoints.

points would require them to be sorted first, resulting in high performance overheads [6].

A solution to this problem is to use a data structure that can still be updated at minimal cost and can return an ordered list of the contained elements, instead of a contiguous log used in HARE. For this purpose HARE++ uses a trie data structure (Figure 3). This data structure is a simple extension of the existing page-table structures used in today's processors [10] for virtual address translation. We extend this structure with an additional fifth level[1], that will store pointers to the checkpointed data for every memory block (64 bytes) within a page. The advantages of using this data structure are: it can easily be updated and traversed by hardware, similar to existing page-tables, and we can generate a sorted list of checkpointed blocks by traversing it in order.

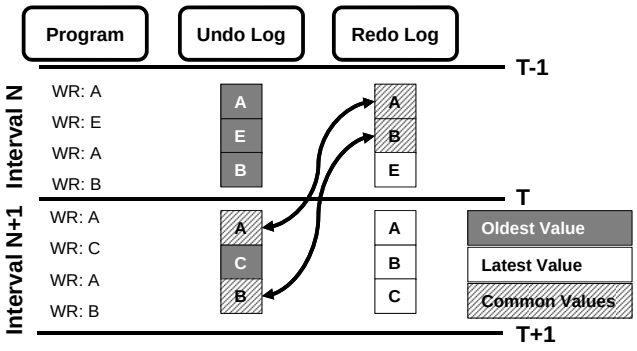

Figure 4. Synergies that develop when both undo and redo-log checkpoints are created.

3.2 Redo-Log Construction

To construct redo-log checkpoints, HARE++ could maintain the original mechanism proposed in HARE [6], but that would increase the overall implementation and performance costs. Instead, when both undo and redo-log checkpoints are being created, the following two properties apply (Figure 4):

- For the same checkpoint interval both the undo and the redo-log checkpoint the same addresses, but they copy different data values: the undo-log copies the oldest data value while the redo-log copies the newest value seen during the checkpoint interval.
- For addresses that appear in both the redo-log checkpoint of interval N (newest values) and the undo-log checkpoint of interval N+1 (oldest values), they checkpoint the same data values.

These two properties allow us to leverage the existing undo-log mechanism and reduce the redo-log construction cost using

[1] The existing page table in x86 64 bit architectures has 4 levels.

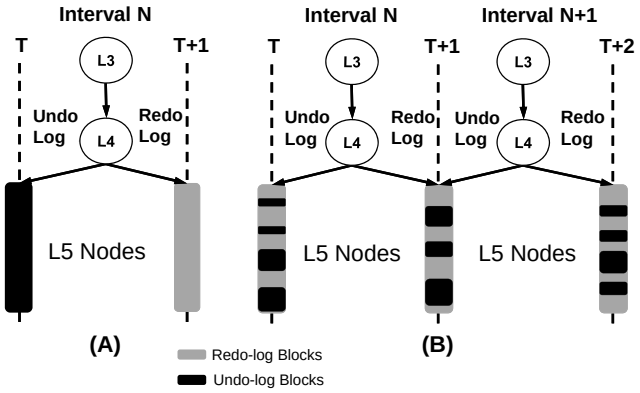

Figure 5. Extended trie data-structure used in HARE++.

the following methods: First, we can use the undo-log meta-data of interval N to identify the blocks to be checkpointed for the redo-log. This approach allows us to remove the redo-log memory modification tracking mechanism used in the original HARE [6]. We eliminate the performance cost of memory tracking, as well as the additional cache design complexity necessary for establishing the atomic read and write-back of both data and meta-data from the L1 to the L2 cache[2]. Second, we can avoid copying the same data value twice, once for the undo and once for the redo-log checkpoint. Instead, we can identify the common blocks and copy them only once when constructing either the undo or the redo-log checkpoint.

To implement both methods we are extending the undo-log meta-data (Figure 3) as shown in Figure 5. First, for every given interval we maintain a single trie meta-data structure that holds both the undo-log and the redo-log meta-data (Figure 5(A)). We implement this by extending the L4 nodes of the trie to have pointers to L5 nodes of both undo and the redo-log checkpoints[3]. The undo-log construction process will update the L5 nodes of the undo-log only, and at redo-log creation time we will walk through the L5 undo-log nodes to find the blocks to be checkpointed by the redo-log.

To identify the common blocks across consecutive redo and undo-logs, we decided the L5 trie nodes of the redo-log meta-data of interval N to be shared with the undo-log of interval $N + 1$ (Figure 5(B)). This solution allows us to delay the construction of redo-log N and not begin it immediately after the end of checkpoint interval N. Instead, we can allow the construction of undo-log N+1 to commence and begin updating the common L5 meta-data nodes with checkpointed blocks. When the redo-log N construction eventually begins, e.g. at the middle of interval N+1, it will not have to copy again blocks already checkpointed by undo-log N+1. This approach also naturally eliminates block *collisions*, which HARE defined as the redo-log blocks that had not been checkpointed yet and were modified by the application[4]. HARE inserted the collision blocks in an unsorted list, and at consolidation time the software was responsible for sorting the collision list.

3.3 Checkpoint Consolidation

Consolidating the checkpoints of two intervals N and N+1 of HARE++ requires the consolidation of both the undo and the redo-logs of the two intervals. For the undo-logs we have to maintain

[2] HARE extends the L1 caches to hold the memory tracking meta-data together with the data, while in L2 the meta-data are cached separately from the data.

[3] The L5 nodes store the pointers to the checkpointed data.

[4] The common blocks across checkpoints are the collision blocks.

the oldest data across the two checkpoints: if both intervals have checkpointed the same address, we maintain the copy of interval *N*, while unique addresses across checkpoints are inserted into the final log. Conversely, when consolidating redo-log checkpoints we have to maintain the newest value of a given address: for the common addresses across intervals we maintain the block checkpointed by interval *N*+1.

The advantage of the original meta-data organization of HARE was that consolidation just required the linear parse of the two lists of addresses that represented the checkpoints. For HARE++, the same process would require (e.g. for the case of redo-logs), for every block of checkpoint *N*, to look-up the trie of interval *N+1* and search if there is an entry, which overall is more expensive than a simple linear list traversal. Two characteristics of HARE++ can accelerate the consolidation process: 1) the L5 meta-data nodes are being shared between consecutive checkpoints, and 2) for a given interval a block marked as redo-log checkpointed implies that there is an undo-log block and vice-versa. Thus, by traversing only the undo-log L5 nodes of interval *N+1*[5] we can consolidate the checkpoints as follows:

- If a block is marked as both a redo-log block of interval *N* and an undo-log block of *N+1*, it can be removed, because there already exists an older undo-log block in interval *N* and a newer redo-log block in interval *N+1*.
- If a block is marked only as undo-logged of interval *N+1* (similarly if it was redo-logged in of interval *N*) then we have to search the undo-log *N* (redo-log *N*+1) for an older (newer) block, and if such a block is found then recycle the block. If no such block is found, insert it in the respective tree.

4. HARE++ Implementation

For constructing the two types of checkpoints and for updating the trie meta-data described in Section 3.1, we use a hardware engine (Figure 6). In the rest of this Section we describe the functionality of the engine and how it interacts with the rest of the system.

4.1 Undo Log Creation

For identifying the memory locations to be inserted into the undo-log, we introduce *checkpoint* bits in the L1 and L2 caches. When a block is written in L1 we check if the checkpoint bit is set, and if not then the block has not been checkpointed in the current undo-log interval: it is sent to the HARE++ engine and we set the checkpoint bit. The checkpoint bit information functions as a first level filtering of blocks to be checkpoint: without them every written block would have to go to the HARE++ engine. When the block is written back to L2 the checkpoint bit follows the data. If a checkpointed block is replaced from L2 we lose the checkpoint bit information, and if this block is written again in the future it is going to be sent again to the HARE++ engine. This mechanism is similar to existing hardware checkpointing techniques such as ReVive [15] and SafetyNet [19]. Duplicate blocks in undo-logs are only a performance and not a correctness concern, as undo-logs are restored in reverse construction order and restored duplicates get overwritten with the correct oldest values. Still, HARE++ performs a secondary filtering of the blocks to be checkpointed by checking if the block is already copied using the trie meta-data, and eliminates any duplicate entries from the final undo-log.

Once a block reaches the HARE++ engine, it is inserted in the *Pending Block Queue* (PBQ). If the PBQ is full then the core sending the block to be checkpointed will have to delay the original write until PBQ has available entries. Once the block reaches the front of the queue it is inserted in the *Tree Construction Engine*, which updates the undo-log part of the trie meta-data for the current checkpoint and checks if the block has already been checkpointed for the current interval N+1. Then the engine checks if the L5 meta-data node of the current undo-log has already been inserted as a L5 redo-log node of the previous checkpointing interval N, and if not it updates the trie meta-data of the previous interval. This operation indirectly inserts the blocks already checkpointed by the current undo-log in the previous redo-log, which is not going to copy them again. The block is then sent to the *Memory Interface* of the engine and written to memory. Once the write acknowledgment is received, the block is marked as *checkpointed* and as an *undo-log* block.

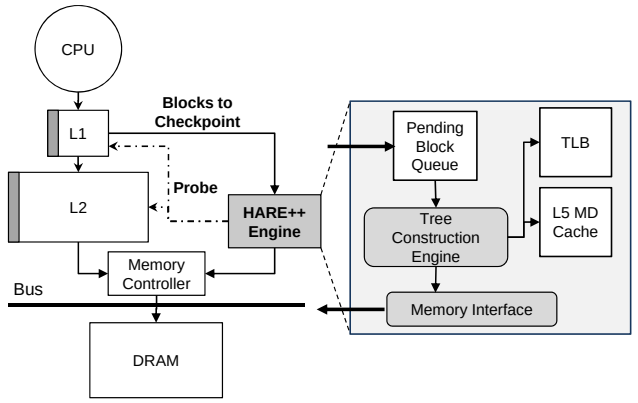

Figure 6. The HARE++ hardware engine.

To efficiently access and update the current and previous trie meta-data structures, the HARE++ engine has a TLB to store the translations for the last level L5 Nodes, and a L5 MD cache to store the contents for the L5 Nodes which are frequently accessed.

4.2 Redo Log Creation

The sets of modified addresses of consecutive checkpointing intervals do not overlap entirely, especially at high checkpointing frequencies where the application does not have sufficient time to modify its working set. For this reason, we have to start actively copying blocks for the redo-log of the previous interval *N* by the end of the current interval *N+1*. HARE++ starts the construction of redo-log *N* in the second half of the current checkpointing interval, e.g. for a checkpointing frequency of 0.1sec, redo-log construction starts after 0.05sec.

To construct the redo-log checkpoint, the *Tree Construction Engine* of HARE++ starts walking the trie meta-data in order, and checks if a block has been copied by the undo-log checkpoint in interval *N*. If yes, then this address has also to be checkpointed by the redo-log checkpoint in the same interval and the block is inserted into the redo-log part of the trie. Then the engine checks if the *L5* node is being shared by the undo-log of interval *N+1*. If the node is not shared, which happens rarely, then it is inserted in the trie of the current undo-log *N+1*. This step is necessary for ensuring that all overlapping *L5* nodes are shared between consecutive tries, and improves performance by eliminating block copies by the undo-log that have already been checkpointed by the redo-log[6]. Finally, the block address to be copied is forwarded to the memory interface that reads and writes the data to memory. The data to be checkpointed can reside in any level of the memory hierarchy and for this reason the HARE++ engine, similar to HARE, has to probe the caches. Once the write acknowledgment arrives, the trie meta-

[5] The same nodes belong to the redo-log of interval *N*.

[6] In such a case the undo-log construction process will mark the block is *u*ndo-log but will not checkpoint it.

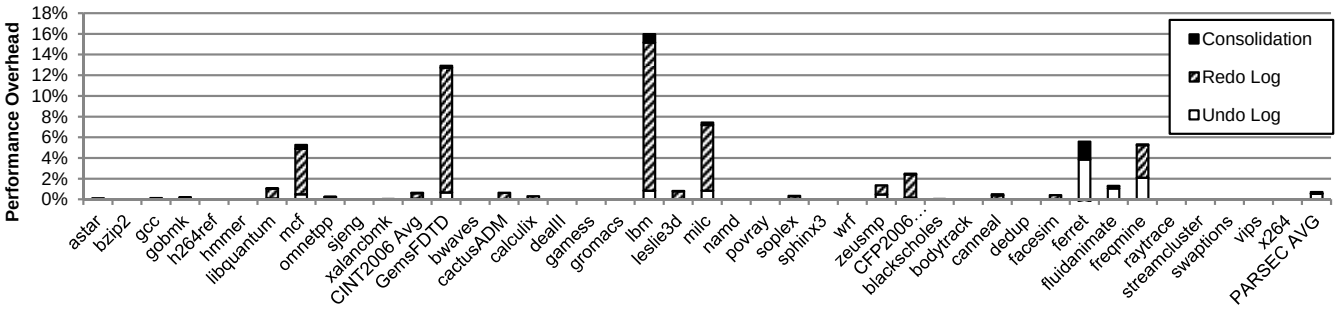

Figure 7. Performance overhead of HARE++ for all simulated applications along with the averages for checkpointing frequency of 0.1 seconds. Breakdown of the performance overhead in: undo-log creation, redo-log creation and checkpoint consolidation costs.

data is updated with the address of the saved block and the block is marked as *redo-log* and *checkpointed*.

4.3 L5 Node Meta-Data Organization

The L5 meta-data nodes store, for every memory block, the pointer to the address where it is being checkpointed, and 3 bits describing whether it is *checkpointed* and if it is part of an *undo* and or a *redo-log*. Since these bits are accessed frequently during undo/redo-log checkpoint creation and consolidation, we improve the data locality of the *L5 MD* cache, by packing the bits of all blocks of the L5 node inside a header (Figure 3). The location of the header within the L5 node is decided based on bits 12-18 of the virtual address, in order to avoid possible address aliasing to the same cache sets and poor *L5* MD cache performance.

4.4 Interaction with the OS/VM

The HARE++ engine is managed by the OS/VM and is equipped with the necessary registers to store, for every currently running process on the processor, the roots of the trie meta-data structures, as well as the core-id of the core where a given process is running. An undo-log block to be checkpointed is accompanied by the id of the core where it was modified and gets inserted in the appropriate tree. The OS/VM is also responsible for providing and managing a list of free memory blocks, similar to the original HARE [6].

5. Evaluation

In our evaluation, we use SESC [16], an open source execution driven simulator, to model a four-core CMP system with Core2-like parameters: 4-issue, out-of-order cores running at 2.93GHz. Each core has a private dual-ported 32KB 8-way associative L1 data cache. All cores share a 4MB, 16-way associative, single-ported L2 cache. The block size is 64 bytes. We model a DDR3-1333-like memory system, which provides $\sim$11.7GB/s and the DRAM average latency is 50ns, which corresponds to 150 cycles. The HARE++ engine we simulate has a 64 entry pending block queue, a 256 entry fully associative trie TLB, a 128KB 16-way associative single-ported L5 MD cache, and the memory interface has a 32 entry read queue and a 32 entry write queue. In our evaluation, we are using the SPEC2006 [21] (Figure 7) benchmark suite, and we simulate 27 out of 29 benchmarks using the reference inputs. We omit perl and tonto because of technical incompatibilities with our simulator infrastructure. In our simulations we fast forward though 5% of the simulation, with a maximum of 20 billion instructions, to skip initialization, then simulate 10 billion instructions. For evaluating multi-threaded applications we use the PARSEC [5] benchmark suite with the *native* input, except in the case of dedup where we use the *simlarge* input, because the application allocates more than 2GB of memory, and exceeds the 32-bit address-space simulated by SESC. For PARSEC benchmarks, we skip to the beginning of the parallel section, then skip an additional 21 billion instructions, and finally we simulate 20 billion instructions. For estimating the final checkpoint memory requirements of our technique, we use PIN [12], where we profiled all SPEC2006 and PARSEC applications to completion, using the reference and native inputs respectively.

HARE++ requires approximately 146KB of on-chip memory state, whereas the original HARE used 4KB, but is still lower than 256KB needed by SafetyNet [19]. We used Cacti 5.3 [22] to estimate the area cost for implementing HARE++, and found that it is 20% smaller than the size of the L1 Data cache, because HARE++'s hardware structures do not need to be optimized for speed as the L1 Data cache does. We also used Cacti to estimate the increase of access latency that the *checkpoint* bit will introduce to the L1 and L2 caches, and we observed no difference compared to the baseline.

Performance Overhead Figure 7 presents the performance overhead of HARE++ for a checkpointing frequency of 0.1 sec. We also show the breakdown of the performance overhead of HARE++ for constructing undo and redo-log checkpoints and consolidating them. HARE++ has an average performance overhead $< 2\%$ across all benchmarks, with the highest overheads being 16% for lbm, $\sim$13% for GemsFDTD and $\sim$8% for milc. In all three cases the primary cause of the overhead is the redo-log creation, whose cost can be explained for the following 3 reasons: 1) all three benchmarks are memory intensive and they create big checkpoints, 2) the HARE++ engine competes with the application for off-chip bandwidth, and 3) only a limited number of redo-log blocks is synergistically checkpointed by the undo-log creation process ($< 10\%$). Undo-log creation in general has low overheads, $< 2\%$ across all benchmarks, with the exception of ferret which is memory intensive. Finally the cost of checkpoint consolidation is $< 2\%$ across all benchmarks.

In our evaluation, we are comparing HARE++ with a hardware and a software scheme. The hardware scheme, labeled *ReVive+HARE*, creates undo-log checkpoints using a hardware technique similar to ReVive [15] and redo-log checkpoints using the original HARE engine implementation (with redo-log consolidation enabled). The software technique, named *TTVM*, is similar to the virtual-machine based checkpointing solution proposed by King et al [18]. In *TTVM* a checkpointing thread executes in parallel with the application and creates both undo and redo-log checkpoints. TTVM uses memory protection and copy-on-write to identify the pages to be copied and checkpoint them. When simulating multi-threaded applications, the checkpointing thread has higher priority and de-schedules the application's threads if no free core is available.

Figure 8 shows the maximum and average performance overhead for the simulated benchmark suites for checkpointing fre-

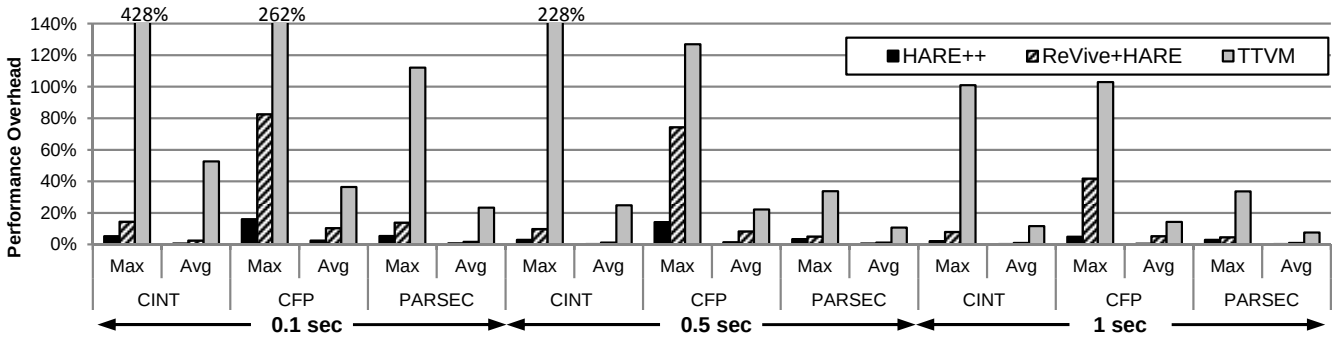

Figure 8. Maximum and average performance overhead of HARE++, compared to ReVive+HARE, and TTVM for checkpointing frequencies of 0.1, 0.5 and 1 seconds.

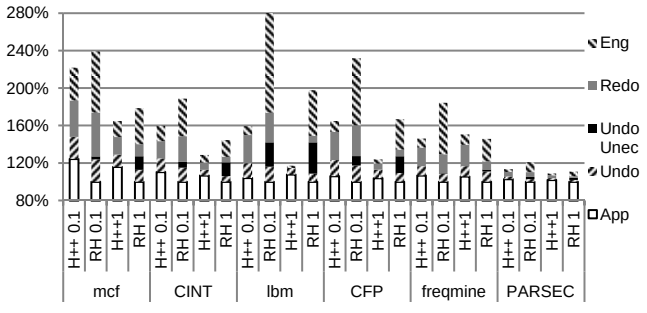

Figure 9. Memory access break down of HARE++ (H++) and ReVive+HARE (RH) for checkpointing frequencies of 0.1 and 1 seconds, for the worst performing benchrmark and the average for each benchmark suite.

quencies of 0.1, 0.5 and 1 second. Both hardware techniques outperform TTVM, which suffers high performance overheads especially at high checkpointing frequencies, e.g. 0.1sec. The primary cause of performance overhead is the pollution of the shared caches with checkpointed data, and the competition between the application and the checkpointing thread for cache space. Another source of overhead for TTVM is the time the application is suspended for servicing the page faults which comprises more than 20% on average (maximum 60%) of the overhead. The higher overhead of ReVive+HARE compared to HARE++ can be attributed to the following reasons: 1) The competition between HARE's checkpoint memory tracking meta-data and application's data in L2, 2) the lack of synergistic copying between undo and redo-logs, which results in the same data being copied twice, and 3) higher consolidation cost, especially for applications which create big checkpoints (e.g. lbm, GemsFDTD), because the size of HARE's checkpoint meta-data is 25% of the checkpointed data.

To gain better insight into the performance benefits of HARE++ compared to *ReVive+Hare*, in Figure 9 we are presenting the breakdown of the average memory accesses per checkpoint interval which consist of: 1) the application memory accesses (App), 2) the necessary undo-log writes (Undo), 3) the unnecessary checkpointed undo-log blocks (Undo Unec), 4) the redo log read and writes (Redo) and 5) rest of the memory access generated by the checkpointing engines (Eng) (e.g. caches misses from HARE++'s L5MD cache, or checkpoint meta-data reads/writes during checkpoint consolidation). The accesses are normalized to the application accesses of ReVive+HARE. The first observation we can make is that HARE++'s engine generates less memory access than ReVive+Hare. HARE++'s L5MD cache proves sufficient for caching the trie meta-data structures during cache checkpoint creation. The merge optimizations also assist, resulting in only 60-70% of trie block pointers being updated on average and reducing the L5MD cache pressure and associated memory accesses. On the other hand, HARE during consolidation has to read the meta-data of both checkpoints, which are lists of addresses/pointers the size of which is 25% of the checkpointed data, and write the resulting consolidated list, resulting in more memory accesses than HARE++. The second improvement is that HARE++ inserts a given address in the undo-log only once, while ReVive may generate duplicate checkpoint entries, as described in Section 4. This phenomenon is especially pronounced in longer checkpointing intervals, e.g. 1sec, where the probability of a block being replaced from the L2 cache, and the associated *checkpoint* bit information to be lost, increases. Finally, synergistic copying can reduce, if not eliminate (e.g. for lbm), the number of blocks copied for constructing the redo-log at low checkpointing frequencies (1sec). In such cases redo-log checkpoints are created practically for free and HARE++ just needs to check the meta-data and verify that the block has been copied.

Checkpoint Memory Requirements HARE++, compared to HARE, checkpoints more data because it creates both undo and redo-log checkpoints and some memory addresses are not shared between consecutive checkpoints. The level of sharing depends on the application's memory access behavior and the checkpointing frequency. At high frequencies (e.g. 0.1 sec) the overlap between consecutive checkpoints is lower, especially for applications which allocate a lot of memory and the memory update period is higher than the checkpointing interval (e.g. GemsFDTD, lbm, mcf). As we consolidate checkpoints, overlap increases: eventually consolidated checkpoints contain almost the entire address space of the application.

Figure 10 presents the maximum and average memory requirements for all benchmark suites, for frequencies of 0.1 and 1 sec, for HARE++. HARE++ has similar memory requirements as HARE, which can be explained by increasing overlap among consolidated checkpoints in HARE++: practically all copied blocks belong to both a redo and an undo-log checkpoint. Both HARE and HARE++ have one to two orders of magnitude less memory requirements than a scheme that does not perform consolidation (please note the logarithmic scale of the graph). To reduce the memory requirements of consolidation-less techniques, lower checkpointing frequencies have to be selected (e.g. 1sec)[7], which will result in increased reverse-execution latency. If consolidation was applied to

[7] The memory requirements of HARE and HARE++ are paradoxically higher at 1sec compared to 0.1 sec, which can be explained by the fewer number of consolidations performed (only one is performed after each checkpoint construction) and the bigger size of the few non-consolidated checkpoints.

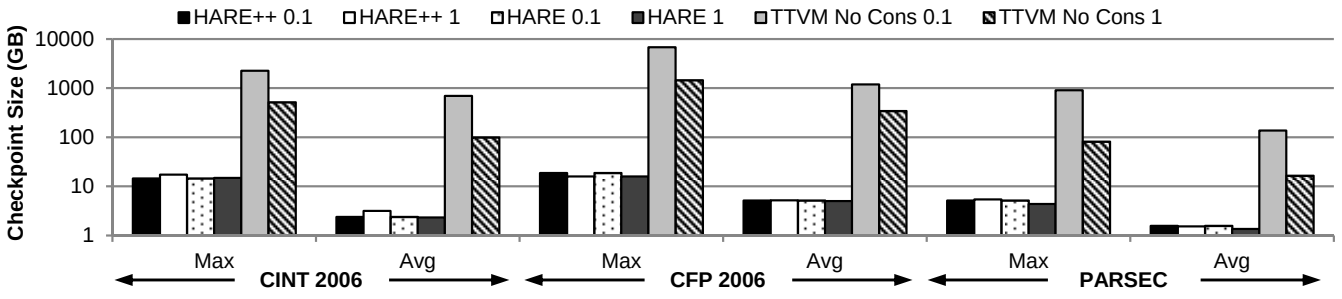

Figure 10. Maximum and average checkpoint memory requirements of HARE++ compared to HARE, and a software technique that does not consolidate checkpoints, for checkpointing frequencies of 0.1 and 1 seconds.

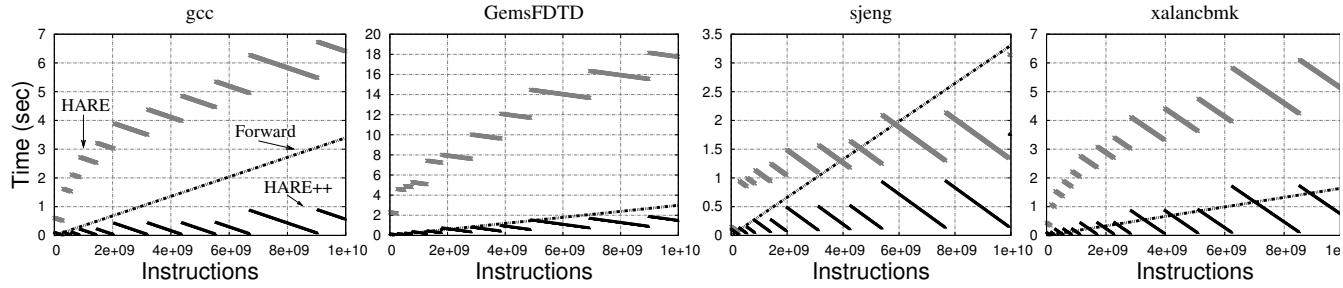

Figure 11. Latency (sec) of reverse executing a number of instruction for HARE++ compared to HARE and forward execution time.

TTVM the memory requirements would be similar (but slightly higher) to HARE and HARE++.

HARE++ has maintained both advantages of the original HARE compared to other techniques: it can deliver the performance benefits of hardware techniques at high checkpointing frequencies and achieve the memory efficiency of software techniques, while creating both undo and redo-log checkpoints.

Reverse Execution Latency The major motivation for improving the original HARE technique was the increased reverse execution latency, because HARE creates redo-log checkpoints which have to be converted to undo-log checkpoints at the beginning of reverse-execution. The conversion processes requires, for a given address in a redo-log checkpoint, a search of previous redo-log checkpoints to find the oldest value. The latency of reverse executing x instructions includes the cost of restoring to the closest checkpoint and deterministically re-executing to the point of interest. We assume that the re-execution latency is a function of the average IPC of the application, as we estimated it in our performance evaluation. Regarding the checkpoint restoration cost, we profiled, using our simulator, the time it takes for both the HARE and HARE++ engines to restore checkpoints of different sizes, and we profiled the average software search time for checkpoints of different sizes, necessary for the redo to undo-log conversion.

Figure 11 shows the time (seconds) that is required for reverse executing x instructions for the first time. The plots show the latency of HARE++ compared to the original HARE and the time it takes to forward execute an equal number of instructions, for the benchmarks with the highest latencies. As we can see, HARE++'s reverse execution time follows closely the forward execution time, because HARE++ just needs to restore the necessary undo-log checkpoint, while HARE suffers higher latencies because of the additional cost of converting the redo-log checkpoints. Figure 12 shows the maximum and average speed-up of reverse execution latency of HARE++ compared to HARE for checkpointing frequencies of 0.1, 0.5 and 1 second. HARE++ can reduce the reverse execution time up to four times on average when we checkpoint at high frequencies (0.1 sec). At high checkpointing frequencies the checkpoint conversion cost of HARE is higher because of lower overlap between consecutive checkpoints which results in more checkpoints being searched. Conversely, at low frequencies we observed that fewer checkpoints are being searched.

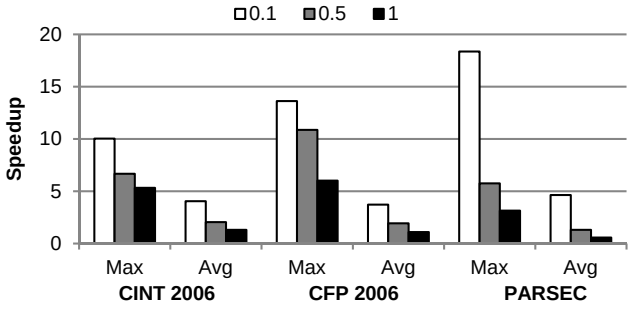

Figure 12. Reverse execution latency speedup of HARE++ compared to HARE.

A solution to HARE's checkpoint conversion cost would be to checkpoint less frequently, but such option would affect directly the reverse execution latency. Figure 13 presents the reverse execution time of HARE++ for checkpointing frequencies of 0.1, 0.5 and 1 second, when reverse executing only a limited number of instructions. Decreasing the checkpointing frequency increases the number of instructions we have to re-execute at reverse execution time, resulting in higher latencies.

Overall, HARE++ has further reduced the reverse-execution time, rendering it directly comparable to forward execution time and improving the interactivity of our technique.

6. Conclusions and Future Work.

In this paper we described HARE++, a redesign of the original HARE technique. HARE++ supports consolidatable undo-log checkpoints which improve the *reverse* execution latency by four times on average. HARE++ also provides the functionality to create redo-log checkpoints as well, by efficiently exploiting the synergies

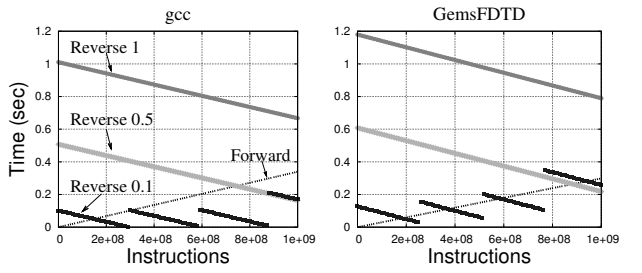

Figure 13. Forward and Reverse execution time of HARE++ for checkpointing frequencies of 0.1, 0.5 and 1 seconds.

that develop when both undo and redo-log checkpoints are being created. These synergies allow HARE++ to eliminate the need for a second memory tracking mechanism for redo-log checkpoints, as well as exploit already checkpointed data and avoid copying them multiple times. These optimizations allow HARE to deliver performance overheads $< 2\%$ on average across all benchmarks and outperform combinations of HARE with undo-log checkpointing techniques which would deliver the same functionality as HARE++. HARE++ preserves the checkpoint consolidation functionality of HARE and delivers the same checkpoint memory requirements. Finally, HARE++ introduces minor modifications to the caches (a single *checkpoint bit*), and efficiently creates checkpoints using a hardware accelerator.

As part of our on-going work we are planning to reduce the hardware requirements of the HARE++ checkpointing engine, by decreasing the size of the used structures (caches, TLB, queues) and study the effects they would have on the performance overhead of checkpointing. Moreover, the trie meta-data structure used by HARE++ provides the opportunity to store more information, e.g. have finer memory tracking granularity, which we can leverage to further improve the bidirectional debugging speed and provided functionality. Finally, we are planning to examine uses of our checkpointing mechanism for improving the reliability and availability of systems.

7. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 0916464, 0964647, and 1017638.

References

[1] B. Boehm and V. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, 2001.

[2] B. Boehm and P. Papaccio. Understanding and Controlling Software Costs. *Soft. Eng., IEEE Trans. on*, 14(10):1462–1477, 1988.

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

[4] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level Checkpointing for Shared Memory Programs. In *11th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, page 235, 2004.

[5] 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.

[6] I. Doudalis and M. Prvulovic. HARE: Hardware Assisted Reverse Execution. In *Proc. of 16th Intl. Symp. on High-Perf. Comp. Arch.*, 2010.

[7] 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.

[8] 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.

[9] S. I. Feldman and C. B. Brown. IGOR: A System For Program Debugging via Reversible Execution. *SIGPLAN Not.*, 24:112–123, 1989.

[10] 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.

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

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

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

[14] 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.

[15] M. Prvulovic 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.

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

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

[18] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *annual conference on USENIX Tech. Conf.*, pages 1–15, 2005.

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

[20] 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.

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

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

[23] 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.