Appears in the Proceedings of the
39th International Symposium on Computer Architecture (ISCA-39), June 2012.

Euripus: A Flexible Unified Hardware Memory Checkpointing Accelerator for Bidirectional-Debugging and Reliability

Ioannis Doudalis
Intel Corporation
ioannis.doudalis@intel.com

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

Abstract

Bidirectional debugging and error recovery have different goals (programmer productivity and system reliability, respectively), yet they both require the ability to roll-back the program or the system to a past state. This rollback functionality is typically implemented using checkpoints that can restore the system/application to a specific point in time. There are several types of checkpoints, and bidirectional debugging and error-recovery use them in different ways. This paper presents Euripus[1]*, a flexible hardware accelerator for memory checkpointing which can create different combinations of checkpoints needed for bidirectional debugging, error recovery, or both. In particular, Euripus is the first hardware technique to provide consolidation-friendly undo-logs (for bidirectional debugging), to allow simultaneous construction of both undo and redo logs, and to support multi-level checkpointing for the needs of error-recovery. Euripus incurs low performance overheads (<5% on average), improves roll-back latency for bidirectional debugging by >30%, and supports rapid multi-level error recovery that allows >95% system efficiency even with very high error rates.*

1 Introduction

The ability to restore the program or the system to a prior state (roll-back) is needed for both bidirectional debugging [3] and error recovery. Roll-back is typically implemented using checkpointing, which records sufficient information to bring the system to a state that it had at the time that the checkpoint was taken. Design of a checkpointing mechanism is usually based on a trade-off analysis between rollback functionality (how far back and how fast can we roll back), implementation cost, memory space overhead, and performance overhead during normal (rollback-free) execution. Frequent checkpointing typically provides better functionality in exchange for more performance overhead, and hardware support [6, 16, 23] has been proposed as a way of dramatically reducing this overhead in exchange for increased hardware cost. To keep this cost low, most prior hardware support mechanisms provide a very narrow range of functionality, so they tend to be applicable for only a limited set of checkpointing uses – some schemes target efficient rollback for bidirectional debugging [3] without considering the needs of error recovery, some provide for error recovery in the short term (for errors that are detected quickly and have not corrupted much memory state) [16, 23], and long-term recovery (from errors that took a while to detect or that corrupted or destroyed a lot of memory content) is usually relegated to a separate (usually software-based) checkpointing mechanism [13]. Unfortunately, if all these mechanisms are implemented in a system, their combined cost, memory space overhead, and performance overhead would be prohibitively high, whereas implementating only one of these mechanisms is hard to justify given the limited functionality that it provides.

This paper presents Euripus, a hardware accelerator for a wide range of checkpointing functionality that can be used for bidirectional debugging and/or error recovery. Euripus exploits the overlap and synergies among different checkpointing needs to reduce the overall hardware cost, memory use, and performance overheads, compared to using a combination of prior mechanisms to achieve similar functionality. Our experimental results indicate that 1) when supporting only reverse execution, Euripus provides lower reverse execution latency than prior (reverse-execution-specific) checkpointing schemes, with similar memory and performance overheads, 2) when supporting error recovery, Euripus provides both long-term and short-term rollback recovery, with performance overheads similar to the most performance-efficient prior schemes (that only supported short-term recovery), and with memory overheads similar to the most memory-efficient prior schemes (software schemes that efficiently support only long-term recov-

[1] At the Euripus strait in Greece the direction of the water flow changes with the tide.

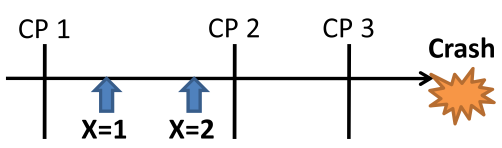


Figure 1. Bidirectional debugging example.

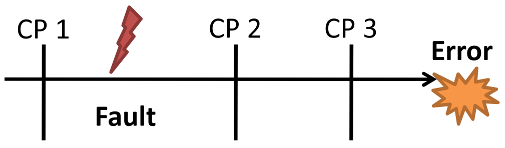


Figure 2. Error recovery example.

ery). Moreover, Euripus can be configured to support both reverse execution and error recovery, with memory and performance overheads only marginally higher than when supporting only one of these uses. Finally, our cost analysis shows that Euripus has a low hardware cost, which is similar to that of prior reverse-execution-only mechanisms.

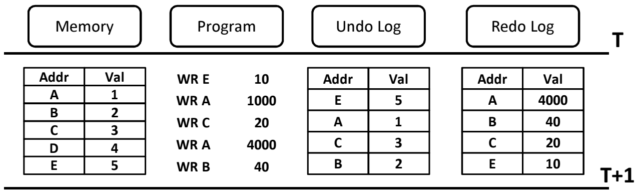
The rest of this paper is organized as follows: Section 2 reviews the checkpointing needs of bidirectional debugging and error recovery and outlines the contributions of this paper, Section 3 is an overview of Euripus, Section 4 presents the implementation of Euripus in more detail, Section 5 presents a quantitative evaluation of Euripus, and Section 6 presents our conclusions.

2 Background and Contributions

Bidirectional Debugging [3] is a promising technique for helping programmers with the daunting and time-consuming task of finding and fixing software bugs, which has been estimated to cause 80% of all software project overruns [11]. Bidirectional debugging lets the programmer examine past states of the program without re-executing the program from the beginning, i.e. it lets the programmer freely move backwards and forward in the execution timeline of the program. The programmer is able, for example, to perform operations like "reverse-step", or set a watch-point on a variable *X* (Figure 1) and then "reverse-continue" to find the last time *X* was modified. This functionality allows the programmer to iteratively, intuitively, and relatively quickly back-track from the effects (e.g. crash) to the cause of the bug, without having to re-execute the program at every back-tracking step. Bidirectional debugging typically provides reverse-execution functionality through a combination of checkpoints and deterministic replay. To illustrate this, Figure 1 shows an execution which ends in a crash *N* instructions after the most recent checkpoint (CP3). A "reverse-step" of *k* instructions from the point of crash would be implemented by restoring the program to CP3, and then deterministically re-executing *N-k* instructions. A "reverse-continue" operation with a watch-point set at variable *X* would be implemented by re-executing past intervals in reverse order (CP3→Crash, then CP2→CP3, then CP1→CP2, etc.) until we find an interval that contains a write to *X*. To verify that this write is the last one to *X*, we execute until the end of this interval, remembering the position of the last write to *X*, and finally re-execute this interval until the correct point (X=2 in our example).

For bidirectional debugging to be useful, it needs to be *interactive*, e.g. "reverse-step" of a single instruction should appear (almost) instantaneous to the user, and a "reverse continue" over some number of instructions should take time that is similar to how long these instructions take to forward-execute. To reduce reverse-execution latency, both of its components must be targeted: 1) deterministic-replay time, which can be reduced through frequent checkpointing [6], and 2) checkpoint restoration time, which can be reduced by reducing the amount of work needed to restore a checkpoint. Unfortunately, frequent checkpointing results in lots of state being copied to checkpoints. Memory requirements of frequent checkpointing can be reduced by "dropping" old checkpoints, but that results in losing the ability to reverse-execute to those past program states. An alternative approach is checkpoint consolidation [3, 6], where checkpoints are progressively merged as they age. Consolidation merges two checkpoints by creating a single checkpoint that contains the union of the addresses saved in the two checkpoints, eliminating (freeing) the duplicates that exist between the two checkpoints. A typical consolidation policy is *exponential consolidation*, where checkpoints which are result of a consolidation are merged again each time their age doubles, resulting in total memory use that is only logarithmically proportional to execution time (without consolidation, memory use grows linearly), while still retaining the ability to roll back to any prior program state with a latency that is proportional to how long ago that state was encountered.

Error Recovery and system reliability are increasingly important, because future systems are expected to be more susceptible to transient [4] and to wear-out-related faults [24]. Worse, processor errors (which can usually benefit from short-term recovery mechanisms) are responsible for only about half of hardware outages (42% [20]), with the rest being attributable to memory errors (orders of magnitude higher than previously estimated [21]), network, software, or the environment (e.g. power failures), etc. To recover from an error (Figure 2), the system should be restored to the last error-free state. When checkpointing is frequent, by the time an error is detected using a low-cost hardware or software error detection technique [8, 19], the latest checkpoint (CP3) may contain a post-error state. Similarly, if the checkpoint is in memory, it may be corrupted by the error (e.g. a memory malfunction or a power failure), and the system must be restored using a checkpoint saved in non-volatile memory [9], e.g disk or NV-RAM. However, non-volatile memory (e.g. disk) has limited write bandwidth, so such checkpoints cannot be created often enough (many times per second) to enable rapid recovery from processor errors.



Memory:

Addr	Val
A	1
B	2
C	3
D	4
E	5

Program:

WR E	10
WR A	1000
WR C	20
WR A	4000
WR B	40

Undo Log:

Addr	Val
E	5
A	1
C	3
B	2

Redo Log:

Addr	Val
A	4000
B	40
C	20
E	10

Figure 3. Undo and redo-log checkpoint examples.

Types of Checkpoints *Full checkpoints* store the entire system/application state and are typically large and expensive to create, while *incremental checkpoints* keep only the modifications during a checkpoint interval. Incremental checkpoints can be either *undo logs* or *redo logs*. Undo logs keep the pre-modification values of modified memory locations (Figure 3), and can be used to roll back the system from a more recent to a less recent state, while redo logs keep the latest values of modified memory locations (Figure 3) and can be used to "fast-forward" the system's state from a less recent to a more recent state. In a hardware implementation, undo logs tend to create less performance overhead because pre-modification values can be saved to the log as modifications are being made, i.e. the writes to the log are spread throughout the checkpoint interval. In contrast, redo logs are constructed after the entire checkpoint interval is executed, because that is when the actual set of modified blocks and their latest values are known. However, consolidation is easier to support in redo logs. In undo logs, data is inserted in order of modification, while consolidation needs to process logs in order of data addresses (to form a union and remove duplicates). Redo logs, on the other hand, are created once the entire set of addresses is known, they are typically created in order of addresses so they are a natural match for efficient consolidation.

Both types of checkpoints (undo and redo log) can be useful. For interactive bidirectional debugging [18], undo logs can provide quick roll-back to past states and redo logs can then provide fast-forwarding to more recent states. Interestingly, undo and redo-log checkpoints can be converted from one type to the other [6], so bidirectional debugging can be supported with only one type of checkpoints – preferably undo logs, because roll-forward functionality can be implemented through re-execution. For error recovery, undo logs can provide quick roll-back from the current to a past state when the error is known to not have corrupted the current state, whereas redo logs can be used to roll forward from a full checkpoint (taken a long time ago) to a more recent state after e.g. a power loss. Redo logs for error recovery are typically created by software, and are stored in non-volatile memory [9] (e.g. disks). Since existing storage media can only provide limited write bandwidth, this limits how often (Figure 5) redo-log checkpoints can be created. This limitation is expected to constrain the scaling of application performance [15], and can be alleviated with the creation of multiple-levels of checkpoints constructed at different frequencies in memory and disk [13] (Figure 5), or with the assistance of novel non-volatile memory technologies, such as PCM [5].

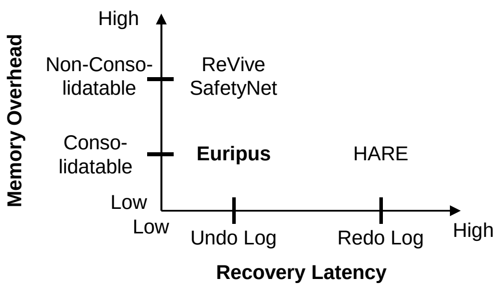


Figure 4. Bidirectional debugging design space.

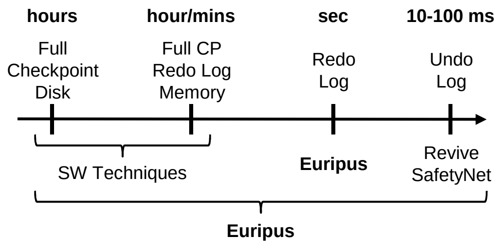


Figure 5. Error recovery design space.

Existing hardware checkpointing techniques provide either non-consolidatable undo logs [16, 23] (Figure 4) for fast recovery from "light" errors, or consolidatable redo logs [6] for bidirectional debugging (with redo to undo log conversion). In future systems, however, several types of checkpoints will be needed, e.g. to efficiently recover from both "light" and "heavy" errors (Figure 5), while possibly also providing reverse execution support for debugging or on-the-fly analysis of malware. Unfortunately, a simple combination of two or more prior techniques would incur unnecessary replication of checkpointing mechanisms, overhead from redundant checkpointing activity, and increased memory requirements.

Contributions Euripus is a new hardware checkpointing accelerator that can provide undo logs, redo logs, or both, constructed with the same or with different checkpointing frequencies, to meet the needs of bidirectional debugging, error recovery, or even both. In particular, the main contributions of this paper are:

- Unlike prior schemes that each populate one particular point in the checkpoint design space (in terms of undo/redo logging, checkpoint frequency, etc.), Euripus can be configured to operate at different points in this design space. Euripus is also the first hardware accelerator that can frequently construct consolidatable undo logs (Figure 4), which allows

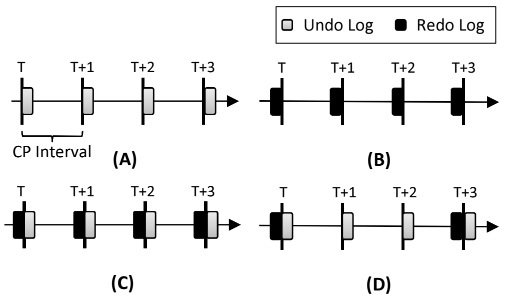


Figure 6. Euripus's modes of operation.

it to e.g. speed up memory recovery in bidirectional debugging by 30% on average compared to prior techniques [6].

- Euripus is also the first hardware technique which can simultaneously construct both undo and redo-log checkpoints, and to exploit synergies between undo and redo logs to avoid unnecessary duplication of hardware, memory space overheads, and performance overheads.

- Euripus is also the first hardware technique that can synergistically create different types of checkpoints at different intervals, e.g. undo logs every Xms and redo logs every Yms, which is important e.g. for enabling both short-term and long-term error recovery.

- Euripus incurs low performance overheads, $<5\%$ on average, by exploiting checkpoint synergies, which enable active memory bandwidth management mechanisms.

3 Euripus Checkpointing Accelerator

Euripus is a flexible unified hardware checkpointing mechanism that aims to provide support for current and future checkpointing requirements of error recovery and bidirectional debugging. Euripus is the first hardware mechanism that can construct consolidatable undo-log checkpoints, which allows it to simultaneously minimize both the memory overhead and reverse execution latency of bidirectional debugging (Figure 4). It is also the first mechanism to provide support for concurrent undo and redo-log construction, either synchronously or asynchronously. Synchronous creation of both undo and redo logs, where both an undo and a redo log is created for the same checkpoint interval, can be used to support both rollback and fast-forwarding in bidirectional debugging. Asynchronous creation of undo and redo logs, where undo logs are created more frequently than redo logs, can be used to support efficient error recovery from both frequent "light" errors (quick recovery using undo logs) and "heavy" errors (long-term recovery using redo logs stored in non-volatile memory). Furthermore, checkpoints created by Euripus are consolidatable, so additional levels of redo-log checkpoints (for long-term recovery) are constructed with little additional cost, which allows

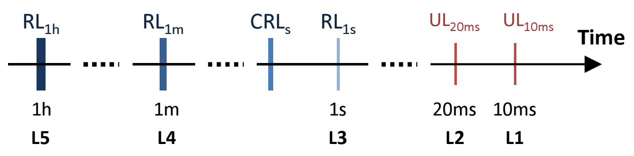


Figure 7. Multi-level checkpointing error-recovery.

Euripus to provide efficient recovery over a wide range of error detection latencies (Figure 5).

It should be noted here that Euripus is designed to efficiently support checkpointing, and that full bidirectional debugging and error recovery support also requires a mechanism for deterministic replay and a mechanism for error detection, respectively. Checkpointing in Euripus is orthogonal to these additional mechanisms, and Euripus can be combined with existing thread race-recording techniques (for deterministic replay) and with existing error detection techniques (for error recovery). Because of its multi-level checkpointing support, Euripus is especially well suited for use with a combination of low-cost error detectors (each for a specific class of errors) that may have widely varying error detection latencies.

Our proof-of-concept Euripus accelerator can provide four specific modes of operation (Figure 6):

- **Undo Logs Only** (Figure 6(A)), which can be used e.g. to support interactive reverse execution.

- **Redo Logs Only** (Figure 6(B)), which can be used e.g. to support less efficient reverse execution, similar to HARE [6], less efficient error recovery (no fast recovery from "light" errors), or both at the same time.

- **Synchronous Undo/Redo Logs** (Figure 6(C)), which can be used in bidirectional debugging for both reverse execution and fast-forwarding, but at a somewhat increased memory space and performance overhead.

- **Asynchronous Undo/Redo Logs** (Figure 6(D)), where undo logs are created very frequently (e.g. every 10ms, for fast recovery from frequent "light" errors) and redo logs are created less frequently (e.g. every 1s, for recovery from "heavy" errors) and then consolidated to provide multi-level checkpointing for error recovery (Figure 7). Undo-log checkpoint consolidation is not desirable in this mode, because frequent consolidations would increase the performance cost, and consolidated undo logs cannot recover "heavy" errors. Reverse execution in this mode can also be supported: use undo logs to reverse-execute only within the supported undo-log window (e.g. 30ms) and use redo log for back-tracking further in the past, which is more efficient compared to the Redo Logs Only mode.

Please note that Euripus has no inherent constrains in terms of implementing other checkpointing strategies (e.g. with different checkpointing frequencies, different Undo/Redo log combinations, which type of checkpoint is stored in which type of memory, etc.) but, given the length limitation for this paper, we limit the discussion mostly to strategies suitable for bidirectional debugging and error recovery, i.e. we will mostly focus on checkpointing frequencies and checkpoint type combinations that are suitable for bidirectional debugging and error recovery. It should also be noted that Euripus is agnostic to the type of memory being used, DRAM or PCM, but because use of non-volatile memory for redo-log checkpoints is important for error recovery, our evaluation provides results on how well Euripus tolerates increased write latencies that are typical among non-volatile memory technologies.

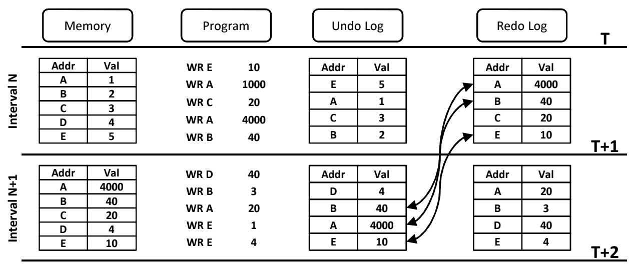


Figure 8. Undo and redo-log checkpoint synergies.

When both undo and redo-log checkpoints are being constructed, either synchronously or asynchronously, Euripus exploits the synergies that develop (Figure 8). In particular, the undo log and the redo log for a given interval contain different data for the same set of modified memory locations: undo logs contain data values from before the first modification (i.e. the value as it was at the start of the interval), while redo logs contain data values from after the last modification in the interval (i.e. the value as it was at the end of the interval). This means that 1) if we have a mechanism that creates a set of addresses to copy to one type of checkpoint (e.g. undo log), a separate modification-tracking mechanism for the other (redo log) is not needed, and 2) if the same address appears in both the redo log for interval *N* and the undo-log checkpoint for interval *N+1*, the data value that should be copied into these two logs is exactly the same, so one copy will suffice for both.

Euripus takes advantage of these synergies. First, it uses its undo-log meta-data as a modification-tracking mechanism for the redo log. At the end of the interval, the undo log contains exactly the set of addresses that should be copied to the redo log. This approach eliminates the hardware and performance cost of a separate redo-log memory tracking mechanism, such as a bit-array used in HARE [6] for this purpose. Second, because an undo and a redo log in neighboring intervals store the save values for any block they have in common, Euripus constructs redo logs in a "lazy" fashion: instead of copying all the values to the redo log at the end of a checkpoint interval, Euripus waits for undo logging activity from the next interval(s) to copy those values to checkpoint memory. In particular, when a block is to be copied to the undo log for the current interval, Euripus also checks if the block is needed in the redo log of the previous interval and, if this is true, inserts a pointer to the copied data to the redo log. This dramatically reduces the memory bandwidth needed for checkpointing, and reduces the bursty memory access pattern that usually plagues redo logging (when all of the copying is done right at the end of a checkpoint interval). Because some locations are modified in one interval but not in subsequent one(s), Euripus eventually goes through each redo log to find addresses which still have no data copied (meaning that the memory location has not been modified since that checkpoint interval), and copies those data values to the redo log.

4 Implementation Details

Euripus is implemented as a hardware accelerator (Figure 11) which is responsible for both copying data to checkpoints and managing the checkpoint meta-data. This accelerator is designed to be implemented at the processor-memory interface (e.g. the on-chip memory controller), to require few modifications to the rest of the processor, and to need few software interventions. This Section describes the structure of our checkpoint meta-data, the undo-log and redo-log construction mechanisms, and the internal structure of the Euripus accelerator.

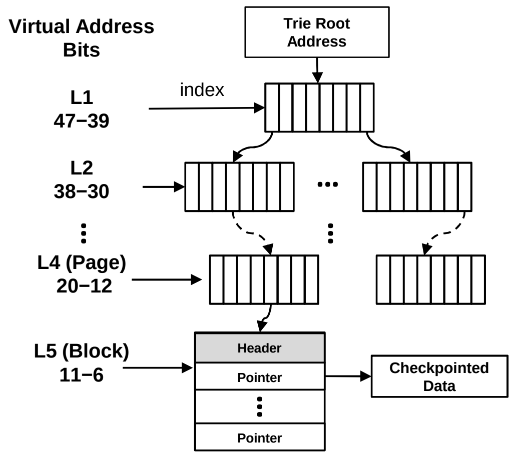


Figure 9. Checkpoint Trie meta-data.

4.1 Checkpoint Meta-Data

The Euripus accelerator can select between two types of meta-data: a contiguous array of checkpointed blocks, or an address-indexed trie (Figure 9) (an extension of hierarchical page tables [10]). In both structures, entries are added as modifications are encountered (i.e. when we find out that a block will be a part of the checkpoint). The contiguous array structure has better locality and is more compact, so it results in less performance overhead than when using the trie structure. However, the trie can be searched by address, which allows efficient consolidation. In principle, any other address-searchable meta-data structure can be used, e.g. a hash table. We decided to use a trie simply because efficient hardware mechanisms already exist for looking up, traversing, and caching such structures.

Like prior hardware checkpointing schemes [6, 16, 23], Euripus checkpoints memory at the granularity of cache blocks. Therefore, the trie structure extends the page-table hierarchy with an additional level that, for a given page, stores the pointers and meta-data bits for each checkpointed block. The meta-data bits specify the type of checkpoint a block belongs to (undo and/or redo log) and if it has or needs to be copied. Figure 9 shows such a trie, where page table levels L1-L4 are similar to an existing page table, and L5 nodes keep Euripus meta-data. The meta-data block bits are accessed much more frequently than the pointers, and are all stored together in a "header" to improve the locality in the accelerator's caches (Figure 12).

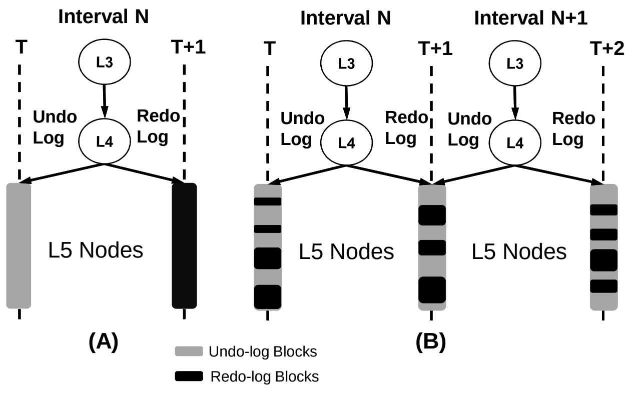


Figure 10. Extended trie meta-data.

Figure 10(A) shows the trie structure when operating in the Synchronous Undo/Redo Logs mode: the undo and redo logs in a checkpoint interval share trie nodes up to the page level (L1-L4), but have separate L5 nodes (shown as gray for undo log and black for redo log) because the data blocks they must point to are different. However, L5 nodes are shared between tries from consecutive intervals, i.e. the redo-log L5 node for interval *N* is also the undo-log L5 node for interval *N+1* (Figure 10(B)). This organization facilitates lazy copying of redo-log blocks: a block that is copied into the undo log for interval *N+1* also ends up in the redo log of interval *N* if the meta-data bits for the block in that L5 node indicate that the block is needed in that redo log.

Euripus consolidates two checkpoints *N* and *N+1* by walking the two tries in address order. If a trie node is present in only one of the tries, it is simply linked into the consolidated trie. When both tries contain nodes for the same address range, one of the nodes is freed after its contents are merged into the other. For undo logs, we keep the node from checkpoint *N* and add to it pointers from *N+1* that were not present in *N*. This results in a consolidated undo-log checkpoint, i.e. one that keeps, for each block modified in the consolidated checkpoint interval, the value it had at the start of that interval. Conversely, consolidation of redo logs retains the nodes from checkpoint *N+1* and merges pointers from *N* that were not present in *N+1*. Note that consolidation does not move saved data block values – they stay in place and only the pointers in the trie are updated. To consolidate synchronous undo/redo logs, we consolidate the (unified undo/redo) L1-L4 nodes, then separately consolidate the L5 undo-log nodes and the L5 redo-log nodes.

Consolidation frees blocks in a non-contiguous fashion, so Euripus maintains a free-list of such blocks and reuses them to save data for new checkpoints. When non-volatile memory (e.g. PCM) is used for checkpoints, the free list is a FIFO to help spread writes among the blocks.

4.2 Undo-Log Checkpointing

When a processor core writes to a data block, it sends the block's address to the Euripus accelerator. To avoid overloading the accelerator with such requests, caches are extended with an extra per-block *checkpoint* bit, which is set when the block is sent to Euripus to indicate that the block has already been checkpointed in the current interval and need not be sent to the accelerator again. These bits are bulk-cleared when a new checkpoint interval begins, are only implemented in on-chip caches, and are initialized to zero when a block is fetched from memory. This initial filtering in Euripus is similar to how prior hardware checkpointing techniques, e.g. ReVive [16], filter writes to their checkpoint logs.

While prior schemes only had this primary filtering, Euripus uses its trie meta-data structure as a secondary filter – when the L5 meta-data indicates that the block has already been checkpointed, it is not checkpointed again. This secondary filtering is precise, so a Euripus checkpoint never contains redundant entries for a block. Unlike the *Undo Log Only* or *Synchronous* mode, where an undo-log trie is constructed, in the *Asynchronous* mode the undo logs are stored in a list. Still, in the *Asynchronous* mode a trie is updated for tracking the blocks to be checkpointed by the redo log, which allows us to check if an undo-block has been checkpointed in the current interval[2].

For error-recovery, memory must contain the latest state of the system at the end of the undo-log interval, so dirty contents of caches have to be written back to complete the undo-log checkpoint. Euripus follows a delayed cache flush

[2]In *Asynchronous* mode, a redo-log interval contains multiple undo-log ones, so L5 trie nodes keep, for each block, the number of the undo-log interval the block was last checkpointed.

approach (similar to Rebound [1]) and writes back only dirty blocks that have not been checkpointed already during the latest undo-log interval; already-checkpointed blocks need not be written back because they will be over-written at recovery time. Delayed flush allows the system to continue executing while dirty blocks are written back, especially from large shared on-chip caches.

4.3 Redo-Log Checkpointing

To construct redo logs, Euripus exploits the synergies between undo and redo logs (see Section 3). In Synchronous Undo/Redo Logs mode, the trie constructed for undo logging already contains the L1-L4 nodes for redo logging. At the end of the checkpoint interval, a software handler finds all (undo log) L5 nodes in this trie and creates the corresponding L5 redo log nodes. It then constructs L1-L4 nodes for the next interval, and inserts these new redo log L5 nodes as undo-log nodes of the new trie. This creates the shared tree structure described in Section 4.1. The actual memory blocks are then copied into the redo log lazily – copying into the new undo log also populates the old redo log (as the two share the same L5 nodes).

In Redo Log Only and Asynchronous Undo/Redo Logs modes, the redo log creation process is nearly identical to this, except that no undo log L5 nodes are created and no undo-log-only copying is performed – writes from processor cores only result in 1) creating placeholder redo-log entries for the current interval, and 2) copying into the redo log for the previous interval, if the meta-data indicates that the block was modified in that interval.

It is possible (and highly likely) that the sets of modified address of two consecutive checkpoint intervals, *N* and *N+1*, are not exactly the same. This means that lazy copying into redo logs will likely leave some redo-log entries without actual saved data. Therefore, at some point Euripus must traverse the redo-log trie and copy data for still-unsaved blocks. To avoid complex searches through many checkpoint tries, Euripus starts this active copying for interval *N* in time for it to complete by the end of interval *N+1*. The question of when to start this copying is an interesting one – starting active copying too early creates a burst of copying activity that might have been avoided if we waited a bit longer for lazy copying to do more work, but starting active copying too late might result in having to stop at the end of the checkpoint interval and wait for active copying to complete. In light of this dilemma, Euripus uses an adaptive approach to throttle active copying, taking into account the current rate of lazy copying, the current (throttled) rate of active copying, and the maximum available (unthrottled) rate of active copying. The current rates of lazy and active copying are obtained by counting how many blocks have been copied to the redo log over a given period of time (e.g. 1000 cycles). The maximum rate of active copying is determined by periodically creating a checkpoint using active copying at full (unthrottled) speed. Armed with this information, Euripus estimates when redo-log construction will be done if the current copying rate (sum of current lazy and active rate) is maintained. It then adjusts the rate of active copying if the estimated completion is too early (more than 10% of the interval will be left) or too late (won't complete by the end of the interval).

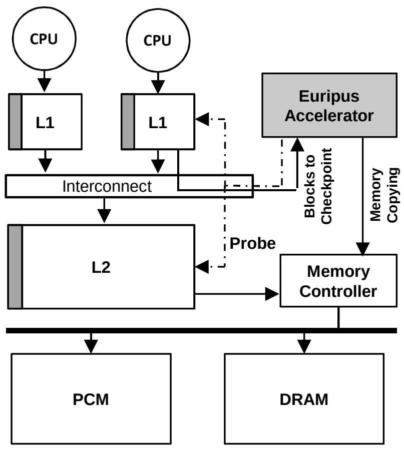


Figure 11. The Euripus hardware architecture

4.4 Accelerator Implementation

The Euripus accelerator (Figures 11 and 12) is positioned close to (or inside) the on-chip memory controller. A single chip can have multiple Euripus accelerators, e.g. one per memory channel or one per N cores, depending on the available memory bandwidth and/or the checkpointing traffic generated by each core. In our initial implementation, we model only one accelerator for the entire multi-core chip. The accelerator receives the blocks to be checkpointed from L1 caches of the cores[3]. Depending on whether checkpointing is done for an application or the entire system, checkpointing can use virtual or physical addresses, both of which are available at the L1 cache level (which are typically virtually indexed but physically tagged).

The Euripus accelerator can support checkpointing of: 1) individual threads/processes, e.g. to support creation of coordinated checkpoints, 2) multi-threaded processes, if we want to checkpoint a specific application, and 3) the entire system. Each of these can be provided by appropriately configuring the accelerator's registers which map the cores, using their core-id, to checkpoint meta-data structures (root pointers of checkpoint tries or current-position pointers for list of addresses). When a block's address is sent to the accelerator, it is accompanied by the core's ID, so the block

[3]In a directory-based protocol, this responsibility can be shifted from the writer's L1 cache to the block's home node, as in ReVive [16].

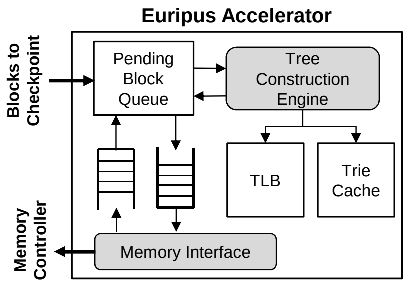


Figure 12. The Euripus accelerator.

can be inserted into the correct trie. When checkpointing each core/thread separately, each core ID has a different meta-data root/current pointer. For process checkpointing, cores that run threads in a process would be mapped to the same meta-data, and whole-system checkpointing is accomplished by mapping all core IDs to the same meta-data.

Once a core sends the block to the Euripus accelerator, it is inserted in the accelerator's pending queue. If the queue is full, which happens rarely, the write on the core has to be delayed. Blocks in the pending queue are first processed by the *Tree Construction Engine* (TCE), which updates the trie to add the corresponding nodes (if needed) and checks meta-data bits to prevent redundant copying (if the block's data is already present in the trie). Blocks that still need copying then go to the memory interface, which saves the block to either DRAM or PCM (depending on which address range is given to Euripus as the checkpoint location), by issuing requests to the memory interface.

The accelerator also monitors how many redo-log blocks still need to be checkpointed for the previous interval, walks the meta-data, and actively copies such blocks. The data to be checkpointed can still reside in the on-chip caches, so the Euripus accelerator behaves like a core when requesting data for copying – it issues cache-coherent requests to get the most recent data from either on-chip caches or (if the data is not in on-chip caches) from memory.

Because Euripus looks up and updates trie meta-data frequently, it uses a TLB to quickly map addresses to L5 trie nodes (i.e. the TLB caches L1-L4 trie look-ups), and a small Trie Cache for keeping the headers (meta-data bits) of the last level (L5 in our examples) nodes of the trie.

4.5 OS, I/O and Multiprocessor Issues

System interactions and I/O are treated differently for bidirectional debugging and for reliability. In bidirectional debugging, each system interaction must be recorded into a *system log* to enable deterministic replay. Euripus can assist the creation of this system log by tracking which memory blocks are modified during a system call: when a core sends a block to the accelerator for copying, it can also forward a system/user mode flag. The accelerator processes user-mode accesses normally, but simply marks (without copying) system-mode accesses as *system modified*, using an additional meta-data bit. At the end of the system call, a (software) search of the checkpoint meta-data can find these *system modified* blocks and save them to a separate system log. Note that system logs must be kept separate from checkpoint data – each individual system call must be replayed deterministically, so a separate copy of *system modified* data is needed for each system call. In contrast, a checkpoint only needs one copy of modified data for an entire checkpoint interval (which can contain many system calls, especially after several consolidations).

For error-recovery, I/O introduces the *output-commit problem* [7], i.e. how to "undo" externally observable actions that occurred between the checkpoint and the point where rollback was initiated. Euripus can be used with existing solutions and workarounds for I/O commit, e.g. ReViveI/O [14] which delays I/O until the end of each checkpointing interval. Frequent checkpointing helps such approaches by reducing the I/O delay time. Another issue related to error recovery with system-level checkpoints is correct handling of DMA and other memory writes initiated outside the processor chip. Like prior schemes, we assume these writes create coherence invalidations, which would trigger normal checkpointing activity in Euripus.

5 Evaluation

To evaluate Euripus, we use SESC [17], an open source cycle-accurate simulator, to model a 4-core processor chip with a DDR3-800 on-chip memory controller (and a detailed DRAM model for the off-chip memory). The cores are 4-issue, out-of-order, with Core2-like parameters, and clocked at 2.93GHz. Each core has a private dual-ported 32KB 8-way set-associative L1 data cache, and all cores share a 4MB, 16-way associative, single-ported L2 cache. Block size is 64 bytes in all caches. In error recovery experiments, we also model an additional memory channel with PCM memory, with an average read latency of 150ns and write latency of 450ns [22]. The Euripus accelerator we model has a 64-entry pending block queue, a 256-entry fully associative TLB, and a 16KB 16-way associative single-ported Trie Cache. Its memory interface has a 32 entry read queue, a 128 entry write queue, and is connected to the on-chip memory controller. In total, the simulated Euripus accelerator is implemented using only $\sim$34KB of on-chip state. Because this state is kept in area-optimized, single ported arrays, its area is approximately 30% the area of a single core's L1 cache (estimated using CACTI 5.3 [26])

Our evaluation uses reference inputs in 27 of the 29 SPEC 2006 [25] benchmarks, shown in Figure 13 [4]. We

[4]We only omit tonto and perl because of incompatibilities with our simulation infrastructure.

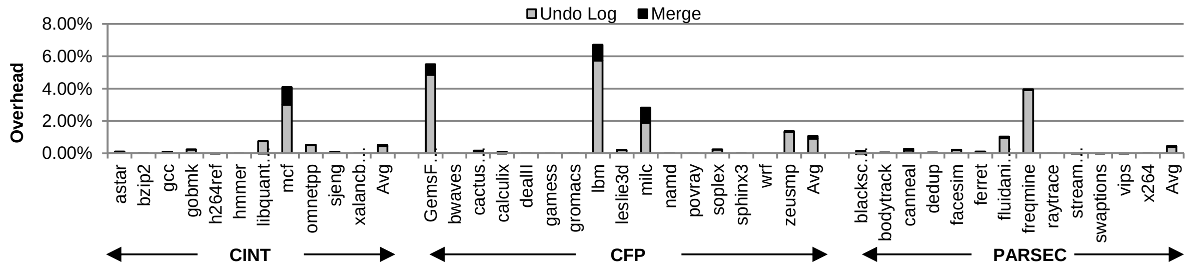


Figure 13. Performance overhead breakdown when constructing consolidatable undo logs.

fast-forward SPEC applications through 5% (up to a maximum of 20 billion instructions) of the execution in order to skip the program's initialization, then simulating 10 billion instructions. We also use all 13 multi-threaded benchmarks from PARSEC 2.1 [2], using native inputs and four threads. The only exception is dedup, where we use the simlarge input – the native input exceeds the simulated 32-bit addresses-space in SESC. We fast-forward PARSEC applications to the beginning of the parallel execution, warm up the checkpointing mechanisms while fast-forwarding over the next 21 billion instructions, and then simulate 20 billion instructions in detail. This number of simulated instructions corresponds to a few seconds (2-5, depending on the application's IPC) of the program's execution.

5.1 Bidirectional Debugging

Figure 13 shows the performance overhead with Euripus, when creating only undo logs every 0.5 seconds, a frequency suitable for interactive reverse-execution. The accelerator consolidates the undo-log checkpoints using an exponential reduction policy [6]. The overall overheads are very low, less than 1% on average and less than 7% worst-case (in lbm). The consolidation overhead is minimal. Euripus performs consolidation in the background without stopping the application, and updates only the meta-data (which is much smaller than the checkpoint's data), so very little contention for memory bandwidth is created. In nearly all applications, most of the overheads are due to the additional memory bandwidth created by data-copying activity during undo-log construction. The only exception is freqmine from PARSEC, where the extra memory bandwidth demand mostly comes from misses in the accelerator's Trie Cache.

Figure 14 compares performance overheads of Euripus, when it constructs only undo logs (Euripus UL) or only redo logs (Euripus RL), with HARE. In all these cases, checkpoints are constructed every 0.5sec and exponentially consolidated. Overall, both Euripus UL and RL have lower performance overheads than HARE, except in freqmine where Euripus suffers from relatively high miss rates in its small Trie Cache. The performance advantage of Euripus on all other applications is due to: 1) Less use of off-chip bandwidth: HARE has to read/write the blocks to

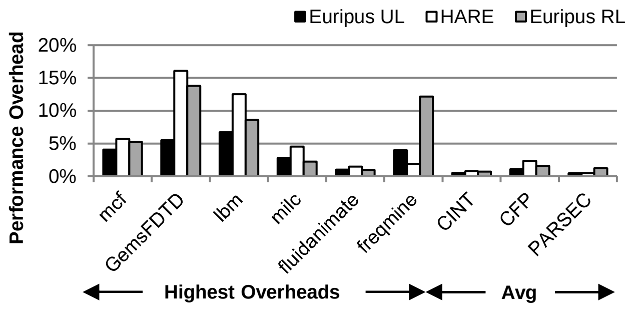


Figure 14. Comparison of Euripus with HARE.

be checkpointed from/to memory to construct the checkpoint, whereas Euripus has zero read memory traffic when constructing undo logs, because it gets the block's values it needs on-chip, 2) HARE suffers from bursty memory access patterns as it creates redo logs (especially in GemsFDTD, lbm, and mcf), while Euripus reduces such behavior – undo logs (Euripus UL) inherently avoid this, while for redo logs (Euripus RL) the memory access pattern is not bursty due to lazy copying and adaptive throttling of the remaining redo-log activity, and 3) Euripus does not need frequent software intervention, which is needed in HARE to generate lists of modified pages or sort its collision lists.

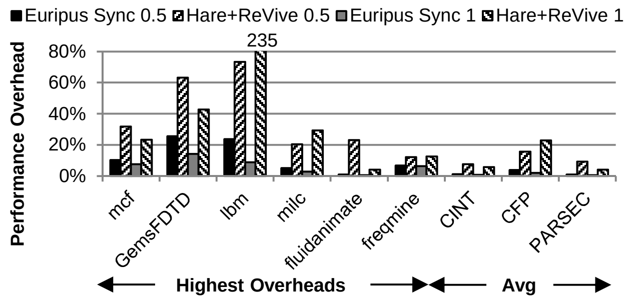


Figure 15. Synchronous checkpoint overhead.

Figure 15 compares Euripus, operating in the Synchronous Undo/Redo Logs mode (Euripus Sync), with a combination of HARE and ReVive that achieves similar (both undo and redo logs) functionality (HARE+ReVive). Euripus incurs performance overheads of <5% on average across all applications. For HARE+Revive, when checkpointing every 0.5s, the cost of constructing the undo and redo logs is similar to Euripus's, and the additional overhead in HARE+ReVive comes mainly from sorting ReVive's undo

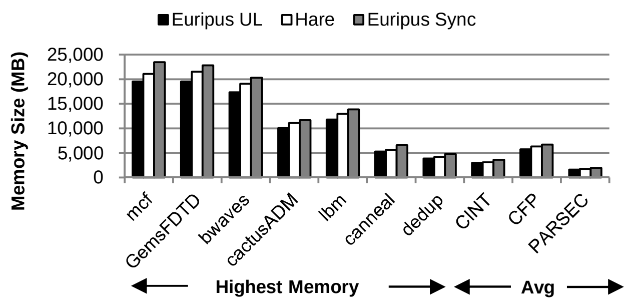


Figure 16. Checkpoint memory requirements.

logs for consolidation (ReVive's efficiency comes at the cost of creating unsorted undo logs, which must be sorted prior to consolidation). At lower checkpointing frequencies (e.g. 1s), Euripus gains additional advantages from 1) synergies between undo and redo logging, and 2) precise filtering that eliminates redundant copying that is present in ReVive (which lacks such a mechanism, resulting in even bigger undo logs e.g. lbm).

To estimate memory requirements of Euripus, we profiled applications using PIN [12] and executed them to completion. We find that memory requirements of Euripus (Figure 16), when constructing only undo logs (Euripus UL) every 0.5sec, are similar to HARE's. In applications that have the highest memory requirements Euripus's memory cost tends to be lower (by up to 1GB) because Euripus's trie meta-data is more space-efficient when representing many blocks than HARE's list-of-addresses meta-data structure. When making both undo and redo logs (Euripus Sync), memory requirements (Figure 16) do not double: Euripus shares checkpointed data blocks between undo and redo logs when a block is modified in consecutive checkpoint intervals. Such blocks are numerous, especially in large consolidated checkpoints that contain most of the application's working set, so construction of both undo and redo logs increases memory requirements only slightly compared to constructing only undo or only redo logs.

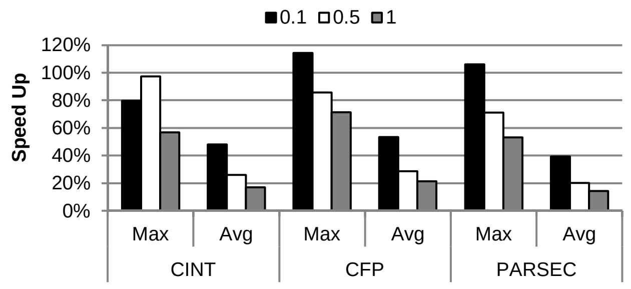


Figure 17. Memory recovery latency speed up.

Finally, Figure 17 shows the maximum and average speed-up of memory recovery latency of Euripus relative to HARE, when back-tracking within the first 2 seconds, for checkpointing intervals of 0.1, 0.5 and 1 second. Euripus can reduce the memory recovery time up to two times on average, when we checkpoint at high frequencies (0.1 sec). At high checkpointing frequencies the checkpoint conversion cost in 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.

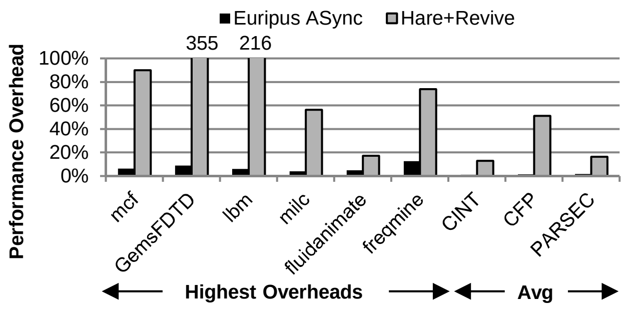


Figure 18. Error-recovery checkpointing overhead.

5.2 Error Recovery

For the purposes of error-recovery we are comparing the asynchronous operating mode of Euripus (Euripus ASync), with a combination of HARE [6] and ReVive [16] (HARE+Revive) that creates checkpoints at the same frequencies: undo logs every 10ms and redo logs every 1s (Figure 18). In this experiment undo logs are not consolidated. Euripus incurs low overheads, less than 2% on average across all applications, while HARE+ReVive's overheads are more than 18% on average, with some cases (GemsFDTD, lbm) having orders of magnitude higher overheads. The primary weakness of HARE+ReVive is that HARE is not designed for a memory subsystem whose components (DRAM, PCM) have different access latencies, and cannot tolerate the high write-latency of PCM. As a result, HARE cannot construct the redo-log checkpoints within a given checkpointing interval, especially for the applications which typically create big checkpoints (e.g. mcf, GemsFDTD, lbm), and often the application's execution has to be paused. Euripus does not suffer from this problem, because the majority of redo-log blocks are lazily copied using undo-log blocks coming from a core and not memory, and the write queue of the accelerator has an increased size, compared to the read queue, in order to support more outstanding writes to PCM.

We also compared the Euripus's adjustive redo-log mechanism to a static one, which starts checking for non-copied redo-log blocks halfway through the interval (e.g at 0.5sec if we checkpoint every 1sec), and copies the redo-log blocks at full speed without performing any throttling. The static policy suffered from additional performance overheads, e.g. 2-3% in applications like GemsFDTD and lbm, because of the reduced number of lazy-copies, and 9% in freqmine caused by the delayed redo-log construction which resulted in pausing the applications execution.

To better understand how Euripus exploits the

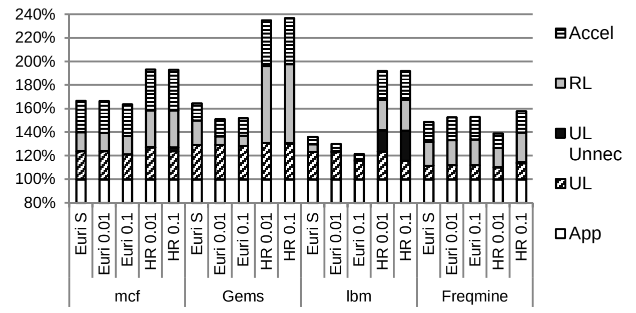


Figure 19. Memory access breakdown.

undo/redo-log synergies and the benefits of the adaptive redo-log policy, Figure 19 presents the memory access breakdown of the 4 applications with the highest overheads. We are comparing Euripus's asynchronous mode using a static policy (Euri S), with one using an adaptive one (Euri). We also compare Euripus with HARE+Revive (HR), when they create undo logs every 10ms (Euri 0.01 and HR 0.01) and every 100ms (Euri 0.1 , and HR 0.1). The memory accesses are broken down to the ones by the application (App), the undo log (UL), the unnecessary undo-log writes (Unnec UL), the redo-log reads and writes (RL), and the ones from the accelerator (Accel). The adaptive redo-log policy eliminates 14% for GemsFDTD and 5% for lbm of additional memory accesses that the static one creates. Euripus requires only 21% for GemsFDTD and 1% for lbm, additional memory (read or write) access for constructing the redo log, while HARE requires 65% and 25% more accesses respectively. On average Euripus needs $\sim$50% for CINT and PARSEC, and 5 times for CFP less reads and writes to construct a redo log than HARE. Euripus can also adapt the undo-log checkpointing frequency and not waste memory bandwidth for duplicate undo-log blocks when we increase the undo-log frequency (e.g. from 10ms to 100ms). The lack of accurate filtering of undo-log blocks causes ReVive to generate for the case of lbm 18% more undo-log writes when the undo-log interval is 10ms, and 26% when it is 100ms. The problem becomes more severe as the undo-log interval increases, and explains the lack of frequency scalability when HARE+ReVive where used for reverse-debugging (Section 5.1). Moreover, Euripus's hardware accelerator consumes less off-chip bandwidth than HARE+ReVive, because Euripus trie checkpoint meta-data are amenable to caching, unlike HARE's list of addresses. Finally, Euripus's bandwidth savings compared to HARE+ReVive not only improve performance, but also translate directly to memory power savings, which is the majority of power consumed in hardware accelerated memory checkpointing.

To estimate the efficiency/availability of a system that uses multi-level checkpointing, we extend the model from Moody*et al.* [13] to support Euripus-like behavior. We then compare the efficiency of Euripus to three systems: 1) one that creates undo-log checkpoints every 10ms but redo logs every every 1 hour (UndoLog+RL1h), 2) another that creates only Euripus's redo-log checkpoints (RedoLog), and 3) one that only creates redo logs every 1 hour (RedoLog 1h). We assume that all redo logs and a full checkpoint are stored in PCM, and obtain checkpoint-restore times from PCM through simulation (1s for a full checkpoint, 1.5 seconds for minutes-level, and 1.75s for seconds-level checkpoint[5]). The base error rate of the system was estimated to be 10^{-8} from field data [13, 20]. This corresponds to about 3 errors per year, and we assume that errors are exponentially distributed through the checkpoint levels[6]. For checkpointing configurations with fewer levels, the errors that correspond to missing checkpoint levels are recovered by the closest checkpoint: e.g. for the UndoLog+RL1h the error rates of the undo-log levels are the same as Euripus's, and the redo-log level's error rate is the sum of the error rates of the redo-log levels of Euripus.

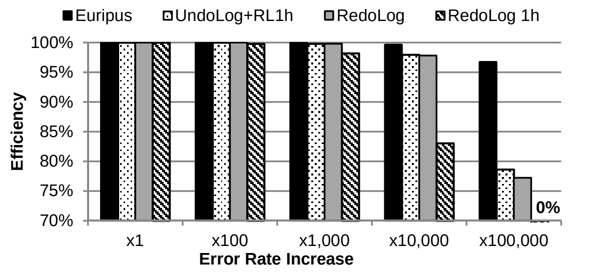


Figure 20. System efficiency for different error rates.

Figure 20 presents the efficiency of the system for Euripus and the other checkpointing configurations for increasing error rates, and can can also be interpreted as the efficiency of an "x" processors system that can create checkpoints and recover at the same latencies as Euripus. Euripus delivers the highest efficiency across all other checkpoint configurations, providing 99.99% efficiency up to 100x error rates, while it can still assist the system achieve availability higher than 95% even when the error-rates increase by 100,000 times. Such high error rates correspond to an error every approximately 15 minutes, which is close to the expected error rate in future exascale systems [20]. The UndoLog+RL1h and RedoLog configurations can support similar availability as Euripus at low error rates, but for the case of extreme error rates their availability would be only $\sim$77%. This decrease in efficiency of both configurations is due to the lack a number of checkpoints that Euripus creates, and does not allow quick recovery when error rates at a specific level increase. Creating checkpoints infrequently, e.g. every 1 hour, has the worst efficiency, because the error frequency is higher than the checkpointing one, and the

[5]Note that rollback to a incremental redo log starts by restoring the previous full checkpoint.

[6]The error rate r_i at level i be $r_i = \alpha \cdot r_{i-1}$, where $\alpha \leq 1$ and $r_{total} = \sum_{i=0}^{l} r_i = \sum_{i=0}^{l} r\alpha^i$

system cannot effectively recover from an error. Finally we performed experiments where we increase the memory recovery latency by 100x, and Euripus can still deliver efficiency at 99%, due the to multiple checkpoint levels which reduce the re-execution time.

6 Conclusion

Bidirectional debugging and error recovery address two separate problems, programmer productivity and system reliability respectively, but both require the same basic functionality: to roll-back the application or the system to a past-state. This functionality is typically provided with the assistance of memory checkpoints, which need to be created frequently in order to achieve interactive bidirectional debugging and high system efficiency.

This paper presented Euripus, a flexible hardware accelerator for checkpointing, which can efficiently support both debugging or error recovery. Euripus can create both undo and redo-log checkpoints, independently or at the same time, consolidate them, and create multiple levels of checkpoints. Euripus exploits the undo/redo-log synergies to reduce performance overhead, memory bandwidth consumption, and memory space requirements of checkpointing, and results in $< 5\%$ average performance overheads, improves the average rollback latency for bidirectional debugging by 30%, and provides error recovery with $\sim$95% system efficiency even at high error rates.

7 Acknowledgments

We thank Nak Hee Seong for helping us understand and model PCM, and Kathryn Morhon, Adam Moody and Greg Bronevetsky for feedback for their analytical multi-level checkpointing reliability model [13]. This work was supported, in part, by National Science Foundation (NSF) grants 0964647, 0916464, and 0903470, and by Semiconductor Research Corporation (SRC) contract 2009-HJ-1977. Any opinions, findings, and conclusions or recommendations in this paper are ours alone and do not necessarily reflect the views of NSF and/or SRC.

References

[1] R. Agarwal et al. Rebound: Scalable Checkpointing for Coherent Shared Memory. In *ISCA-38*, 2011.

[2] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT-17*,2008.

[3] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *PLDI*, 2000.

[4] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 2005.

[5] X. Dong et al. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems. In *SC*, 2009.

[6] I. Doudalis and M. Prvulovic. HARE: Hardware Assisted Reverse Execution. In *HPCA-16*, 2010.

[7] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast output Commit. *IEEE Transactions on Computers*, 1992.

[8] S. Feng et al. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *ASPLOS-15*, 2010.

[9] C. S. Gary L. Mullen-Schultz. *IBM System Blue Gene Solution: Application Development*. 2007.

[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 et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[13] A. T. Moody et al. Detailed Modeling , Design , and Evaluation of a Scalable Multi-level Checkpointing System. *Technical Report*, 2010.

[14] J. Nakano et al. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *HPCA-12*, 2006.

[15] R. Oldfield et al. Modeling the Impact of Checkpoints on Next-Generation Systems. In *MSST-24*, 2007.

[16] M. Prvulovic and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *ISCA-29*, 2002.

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

[18] Samuel T. King et al. Debugging Operating Systems with Time-Traveling Virtual Machines. In *USENIX*, 2005.

[19] S. K. Sastry Hari et al. mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *MICRO-42*, 2009.

[20] B. Schroeder and G. Gibson. A Large Scale Study of Failures in High-Performance-Computing Systems. *IEEE Transactions On Dependable And Secure Computing*, (November), 2009.

[21] B. Schroeder et al. DRAM Errors in theWild: A Large-Scale Field Study. In *SIGMETRICS-11*, 2009.

[22] N. H. Seong et al. Security Refresh: Prevent MaliciousWear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping. In *ISCA-37*, 2010.

[23] D. J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *ISCA-29*, 2002.

[24] J. Srinivasan et al. The Impact of Technology Scaling on Lifetime Reliability. In *DSN*, 2004.

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

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