KIMA: Hybrid Checkpointing for Recovery from a Wide Range of Errors and Detection Latencies

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

Abstract

Full system reliability is a problem that spans multiple levels of the software/hardware stack. The normal execution of a program in a system can be disrupted by multiple factors, ranging from transient errors in a processor and software bugs, to permanent hardware failures and human mistakes. A common method for recovering from such errors is the creation of checkpoints during the execution of the program, allowing the system to restore the program to a previous error-free state and resume execution. Different causes of errors, though, have different occurrence frequencies and detection latencies, requiring the creation of multiple checkpoints at different frequencies in order to maximize the availability of the system.

In this paper we present KIMA, a novel checkpointing creation and management technique that combines efficiently the existing undo-log and redo-log checkpointing approaches, reducing the overall bandwidth requirements to both the memory and the hard disk. KIMA establishes DRAM-based undo-log checkpoints every 10ms, then leverages the undo-log metadata and checkpointed data to establish redo-log checkpoints every 1 second in non-volatile memory (such as PCM). Our results show that KIMA incurs average overheads of less than 1% while enabling efficient recovery from both transient and hard errors that have a variety of detection latencies.

1. Introduction

System reliability is an increasingly challenging problem. Errors can be caused at any level of the system architecture from the processor to the application, and can propagate through the hardware/software stack resulting in erroneous program execution or to an application or system crash. Decreasing device sizes, render processors increasingly vulnerable [3] to errors and failures, e.g. transient failures from particle strikes and intermittent or permanent faults due to wear-out phenomena. The reliability problem is well-known in massively parallel processing (MPP) systems, where the probability of hardware errors is high because it increases with the number of cores of the system. The Mean Time to Failure (MTTF) in a MPP system can be as low as 6.5 hours [21], and future exascale systems are expected to experience an error every 3-26 minutes [25]. Processor errors, either transient or permanent, still comprise only a part of all possible error types a system has to recover from; they are reported to be responsible for only 42% of hardware outages and 24% of total errors [24]. Other common causes of failures are memory errors, followed by network, software or environment (e.g. power failures), etc. For the case of DRAM, for example, a recent study [26] based on field data reports that one third of the machines will suffer a recoverable error, while 1.3% of them will encounter an uncorrectable one; these DRAM error rates are orders of magnitude higher than previously estimated.

To identify and recover from the multiple types of errors, systems can use two approaches: forward error recovery (FER) or backward error recovery (BER). Solutions which rely on FER use resource redundancy [1], which replicates structures of the system (e.g. processor), and execute the same operation (e.g. program) multiple times, using voting to select the final result. Such solutions provide systems with the highest degree of availability, but are very expensive in terms of power, cost and performance overheads. This cost is especially high when many system components (DRAM, buses, etc) are replicated.

The lower-cost BER approach relies on checkpoints/logs to restore execution to a safe point that existed before the occurrence of the error, then restart execution from there. The recovery time of BER is a function of the time to restore the state of the system (processor, memory, etc) and the time necessary to re-execute the program until the point of the error. This recovery time can be minimized by combining frequent checkpointing with low-latency error detection. Low-cost hardware and software techniques [9, 20, 23, 32] detect processor errors by their effects on higher levels of the system, and can identify and recover from an error within several milliseconds of program execution. However, there is a small percent, $< 2\%$, of processor errors [23] which can escape that recoverability window. These errors, such as memory corruption, partial power supply failures, etc. can have much longer detection latencies. Finally, some failures, such as total loss of power, disable the detection and recovery mechanisms along with the rest of the system, so recovery from them requires full-state checkpoints in non-volatile storage. Although long-detection-latency and catastrophic failures represent a small percentage of errors, they result in much longer recovery times, so they still have a major impact on the overall availability and recoverability of the system.

There are two types of memory checkpoints that BER can use, undo-log and redo-log checkpoints. Undo-log checkpoints [18, 28] save the memory state necessary to restore the program from the current point in time to the point of time T in the past. Frequent undo-log checkpoints, combined with low-latency error detection mechanisms [23] can provide low recovery latency. The main disadvantage of undo-log checkpoints is that they require the "current" state of the memory to be preserved in spite of the failure, rendering them unsuit-

able for recovery from hard errors such as DRAM or power failures, or from other errors that may have corrupted the non-checkpointed memory state. Redo-log checkpoints overcome this problem by restoring the system state starting from a full checkpoint (or the beginning of execution), then applying redo-log information to update the system to a more recent state. The disadvantage of redo-logs is that they require periodic creation of a full checkpoint, which includes a copy of the entire memory state of the system, in addition to the redo-log that keeps updates for recent modifications. As a result, redo-log checkpoints have traditionally been created infrequently and stored on hard-disks using high performance I/O infrastructures[34]. In the future, though, as the number of processors increases, the increased checkpointing requirements and the limitations of disk storage technologies are expected to constrain the scaling of application performance [16]. To overcome this problem, multi-level checkpointing schemes [14] have been proposed that improve the efficiency of the system while reducing the I/O load.

In this paper we present KIMA, a hybrid checkpointing mechanism where both undo-log and redo-log checkpoints are created in order to provide the maximum recovery coverage and limit the recovery time from different types of system errors. KIMA has the following characteristics:

- Exploits the synergies between consecutive undo- and redo-log checkpoints, eliminating the need for multiple memory tracking mechanisms and reducing the total performance overhead, by checkpointing memory addresses common across consecutive undo- and redo-logs only once.
- Uses a specialized hardware engine to create checkpoints in parallel with the program execution.
- Creates undo-log checkpoints at high frequencies (e.g. 10ms), for quick recovery from transient errors with short detection latencies, but also establishes incremental redo-log checkpoints less frequently (e.g. every 1s) to enable recovery from hard system errors or errors that escape early detection.
- Employs meta-data structures which enable the consolidation of redo-log checkpoints, creating additional checkpoints at different frequencies (e.g. every minute or hour) for no additional memory copying cost, assisting future multi-level checkpointing techniques.
- Allows the system to adjust the number and frequency of checkpoints depending on its error detection latency characteristics and available memory for storing checkpoints.

Overall, KIMA incurs overheads of $\sim 1\%$ on average even at high checkpointing frequencies (10ms for undo-log and 1sec for redo-log checkpoints), while creating a wave[1] of checkpoints that follow program execution (e.g. at 10ms, 1s, 1 minute, and 1 hour distances), enabling the efficient recovery both from catastrophic failures and from non-catastrophic ones, and across a wide variety of detection latencies (milliseconds to one hour).

[1] Kima means wave in Greek.

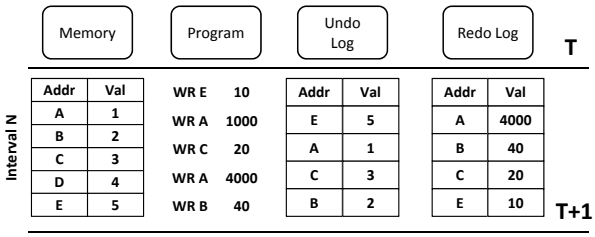

Figure 1. Undo-log and redo-log checkpoint creation.

The rest of this paper reviews existing checkpointing techniques (Section 2), provides an overview of KIMA (Section 3) and discusses KIMA's implementation details (Section 4), then presents our quantitative evaluation (Section 5) and conclusions (Section 6).

2. A Review of Checkpointing Techniques

Checkpointing is typically used by reliability [18, 28] and debugging [8, 33] mechanisms which require to roll-back the program/system to a past state S that existed at time T. Undo-logging uses a log with the old values of the addresses which have been modified between time T and the next checkpoint (or present time). To roll back to state S, undo-logs are applied to the current state in reverse order, effectively undoing changes that were made since time T. The other approach, redo-logging, saves at time T the new values of locations that were modified between the previous checkpoint and time T. To roll back, it restores the state from an older full checkpoint C, then updates it with subsequently created logs until state S is reached. Figure 1 illustrates the two approaches. For a given time interval, between time T and $T+1$, both approaches track memory modifications. For an address which is written for the first time in the current interval (e.g. A), undo-logging copies old data to the undo-log and marks it as checkpointed. Another write to an address which has already been checkpointed (A) does not result in another entry in the undo-log checkpoint. Note that log entries are finally ordered by when data addresses are modified by the program. To create a redo-log checkpoint for the time interval between T and $T+1$, we just note which addresses have been modified by the execution of the program and, at the end of the interval, copy the newest values of the modified addresses. As noted by HARE [8], this approach allows the log to be sorted by address (Figure 1) and allows efficient checkpoint consolidation.

Because rollback with undo-logging begins with the current state of the system, it can only be used if the "current" state is preserved, which limits its scope. In contrast, redo-logging provides recovery even if all of the current state is lost (e.g. by loss of power), but is less efficient because it saves data in bursts and needs to periodically create full-state checkpoints. Note that redo-log checkpoints should be saved to non-volatile memory, so they can be used to recover from catastrophic failures. Conversely, undo-log checkpoints need not be saved to non-volatile memory, because they are rendered useless by any failure that results in losing the current DRAM state.

In terms of implementation, checkpointing can be done at the level of the application [4], runtime library [17], or the operating system [29]. Software implementations typically keep track of the memory modifications at the page granularity, using existing page protection and dirty bit mechanisms. They suffer from significant overheads when checkpointing is done frequently (e.g. every 1sec), primarily because the frequent copying activity competes with the application for processor time, and because frequent checkpointing causes some frequently modified pages to be copied often (once per checkpoint).

Hardware support is necessary for efficient and low overhead high frequency checkpointing. Hardware mechanisms [8, 18, 28] reduce the performance overhead by efficiently tracking the memory modifications at cache block granularity, and by overlapping the program execution with the checkpoint creation. ReVive [18] and SafetyNet [28] create undo-log checkpoints at frequencies from 10ms to 100ms to recover from transient errors in the processors or the system (e.g. lost network packets). Both schemes modify the caches [28] or the coherency protocol [18] to keep track of the checkpointed blocks and log the old data of a block when it is modified for the first time in a checkpoint interval. The main difference between ReVive [18] and SafetyNet [28] is that ReVive stores the checkpoints in memory, while SafetyNet uses only on-chip buffering. As a result ReVive can tolerate longer error detection latencies as well as processor failures, but has higher performance overheads.

To reduce the memory requirements of systems which need to maintain multiple checkpoints, such as bidirectional debugging, software solutions proposed the consolidation of checkpoints [2]. A consolidated checkpoint is the union of the set of addresses of the two input checkpoints, while duplicate entries (addresses common to both checkpoints) maintain the data of the latest checkpoint. A hardware mechanism, HARE [8], has been designed to improve performance of redo-logging and consolidation for debugging uses. Unlike hardware mechanisms which create undo-log checkpoints [18, 28], HARE uses more complex meta-data structures to keep track of modified blocks and, at the end of the checkpoint interval, copy them in parallel with the program execution. Interestingly, HARE restores the program to a past state by first constructing undo-log checkpoints from its redo-log ones, a processes which can be completed within the time constraints of bidirectional debugging, but has much higher latency than restoring a pre-existing undo-log checkpoint.

Another important consideration for checkpointing, especially redo-logging, is persistent storage. Frequent redo-log checkpointing would by far exceed the available bandwidth of disk drives and I/O systems. Multi-level checkpointing [14] has been proposed to reduce the I/O requirements of MMP systems. Multi-level checkpointing caches checkpoints in the local memory of a node of an MMP system, instead of writing them directly to disk, allowing the system to recover 85% of errors using cached checkpoints and reducing the disk I/O load by half. Another technology that can enable frequent redo-logging is phase change memory (PCM), a non-volatile memory technology that is expected to scale beyond the limitations of DRAM in the future [11]. Compared to DRAM, PCM has a higher access latency and a limited number of writes that can endure over its lifetime. However, PCM is far superior to both disks and flash memory. Recent research results have demonstrated how to overcome the limitations of PCM and use it as the main system memory [12, 19], improve its lifetime [27], and construct efficient file-systems [6]. Finally, PCM can improve the scalability of checkpointing schemes in future exascale systems [7].

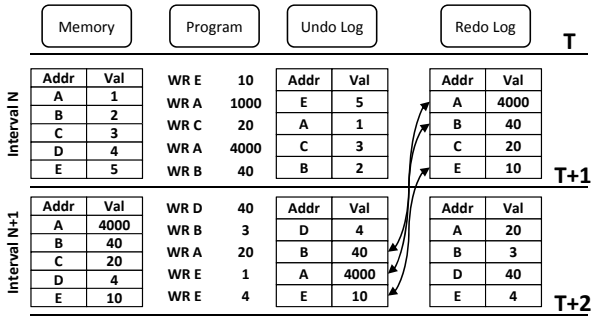

Figure 2. Example of memory locations which are common across consecutive redo-log and undo-log checkpoints.

3. Overview of KIMA

In Figure 2 we show the contents of undo- and redo-log checkpoints for consecutive checkpointing intervals. We can make the following two observations regarding the synergies that develop when both undo- and redo-log checkpoints are being created.

- For a given time interval, the set of modified addresses of both the undo-log and the redo-log are the same; the only difference is that the undo-log saves the oldest data values for each of these addresses, while the redo-log the newest (Figure 2). This means that undo-log meta-data (addresses of saved blocks) can be leveraged by redo-logging to avoid a separate memory-modification tracking scheme.

- If an address (e.g. A) is modified in interval N and in the next interval $N + 1$, the data saved in the undo-log for interval $N + 1$ is the same as the redo-log data for interval N. This property can be leveraged to reduce the amount of copying that is needed to perform undo- and redo-logging together.

KIMA exploits these two observations to create both undo- and redo-log checkpoints efficiently, and to overcome the performance, storage bandwidth, and recovery latency limitations of prior checkpointing schemes. KIMA creates undo-log checkpoints frequently (10ms - 100ms) to provide low recovery latency from early-detected errors. It then exploits the synergy between undo- and redo-logs to efficiently create redo-logs at second-level intervals that enable recovery from more severe errors. Because some error detection latencies can be longer than one second, KIMA consolidates redo-log checkpoints to also create minute- or hour-level redo-log checkpoints. A full checkpoint is also maintained, created at the beginning of the execution and updated by the incremental redo-

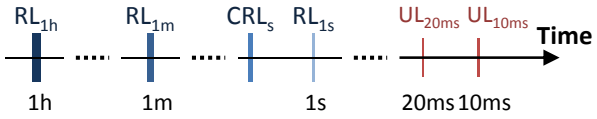

Figure 3. Distribution of undo-logs and redo-logs checkpoints over time.

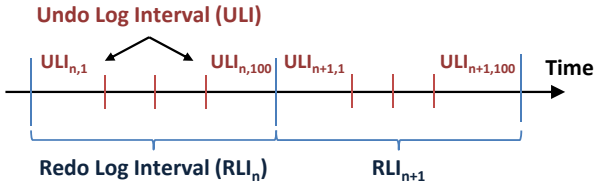

Figure 4. Distribution of undo-log and redo-log intervals over time.

log checkpoints. These redo-log checkpoints allow KIMA to recover quickly from errors that escape the detection latency of mechanisms which rely on undo-log checkpoints [23], or which cannot be recovered from using undo-logs (e.g. because of a DRAM error).

KIMA is a checkpointing solution which is orthogonal to existing error-detection mechanisms and can adjust without incurring significant additional performance overheads to specific checkpointing configurations, as we will show in our evaluation (Section 5). The exact number of undo- and redo-log checkpoints, the frequency at which they are created, and the distribution of incremental redo-log checkpoints over-time are decided by the recovery mechanism [7, 14] based on the following parameters: 1) the expected frequency of errors at every checkpointing level, 2) the resiliency of the checkpointing level to errors (e.g. checkpoints stored in DRAM) 3) the latency of error detection mechanisms, 4) the size, and 5) the available memory of the system. For the purpose of describing and evaluating KIMA we are assuming the frequencies and distribution of checkpoints shown in Figures 3 and 4.

KIMA creates undo-log checkpoints (UL) every 10ms and maintains a limited number of completed checkpoints (e.g. 2), plus one that is under construction; this approach allows KIMA to roll-back the program state by several milli-seconds (e.g. 30) using undo-logs (Figure 3). Undo-log checkpoints older than the selected number are discarded (and their memory freed/recycled). While creating undo-logs, KIMA also tracks modified blocks for the next redo-log checkpoint (which is in the modification-tracking stage at the time) and leverages undo-log data copying to save data to the previous redo-log checkpoint (which is in the data-copying stage at the time). A redo-log checkpoint is completed every second; after a new redo-log checkpoint is created, the previous one is consolidated (similar to HARE [8]) into a consolidated redo-log (CRL) checkpoint for the current minute-long interval. At the end of the minute, the minute-scale CRL is completed and the prior one is then consolidated into the hourly checkpoint for the current hour-long interval. At the end of the hour, the new hour-scale checkpoint is complete and the old one is used to update the full checkpoint, which we assume is stored on disk. To restore program state after a hard failure (or an error whose latency is longer than 30ms), KIMA restores the full checkpoint and then applies hour- , minute-, and second-scale checkpoints until the error latency interval is reached[2]. For example, if the error detection latency for a particular error is one hour or less, KIMA would just restore the full checkpoint. If the detection latency is one second, KIMA restores all but the second-scale checkpoints.

The hierarchy of checkpoints in KIMA also dictates the bandwidth management of memory, PCM and disk. Undo-log checkpoints are created frequently (10-100ms), so they require very high write bandwidth (up to 900MB/s for some applications, Figure 5). This bandwidth can be supported by DRAM memory, where KIMA stores undo-log checkpoints. Redo-log checkpoints created each second still require significant bandwidth (almost 800MB/s for the SPECFP application [30]), but are saved in PCM that can sustain this write bandwidth. Consolidation of redo-logs only requires meta-data changes (no copying of data blocks) and consumes far less bandwidth than data copying. Finally, hour-scale updates are sent to disk, but the bandwidth requirements for this can easily be sustained by modern disk drives – as shown in Figure 5, even if minute-scale checkpoints were sent to disk, the required bandwidth would be only up to 35MB/s.

To create the two types of checkpoints, KIMA uses the following approach: A) For undo-log checkpoints, KIMA keeps track of the checkpointed blocks in a given undo-log interval (ULI) as described in Section 4.1. B) For establishing redo-log checkpoints KIMA takes advantage of the two properties of the undo-log and redo-log checkpoints described in this Section. First, KIMA uses the meta-data of the multiple undo-log checkpoints that are created during a redo-log interval (RLI) (Figure 4), in order to construct the set of modified addresses for the given RLI and create a redo-log checkpoint. This approach eliminates the need for a second hardware assisted modified memory tracking mechanism similar to the one used by HARE [8]. Second, KIMA identifies the block addresses which belong both to the current undo-log intervals ($ULI_{n+1,1}, \ldots, ULI_{n+1,100}$) and to the previous redo-log interval (RLI_n). The first time such block is identified (e.g. in $ULI_{n+1,1}$) it is stored directly to the redo-log checkpoint for RLI_n in PCM, and the undo-log meta-data of $ULI_{n+1,1}$ is updated to point to the same block in PCM. If this address is found again in following ULIs, it is saved to DRAM because it does not represent the newest values for RLI_n. This method eliminates future reads and writes that KIMA would have to perform for creating the redo-log checkpoint of RLI_n, reducing the memory bandwidth requirements.

Another important problem when restoring a system is recovering I/O (e.g. network). KIMA is orthogonal to existing techniques for restoring I/O (e.g. ReViveI/O [15]). The support for frequent undo-log checkpointing and quick recovery from transient errors can reduce the necessary buffering.

[2]To avoid over-writing multiple times addresses common across checkpoints KIMA's meta-data allow to consolidate all incremental redo-log checkpoints first into a single full checkpoint

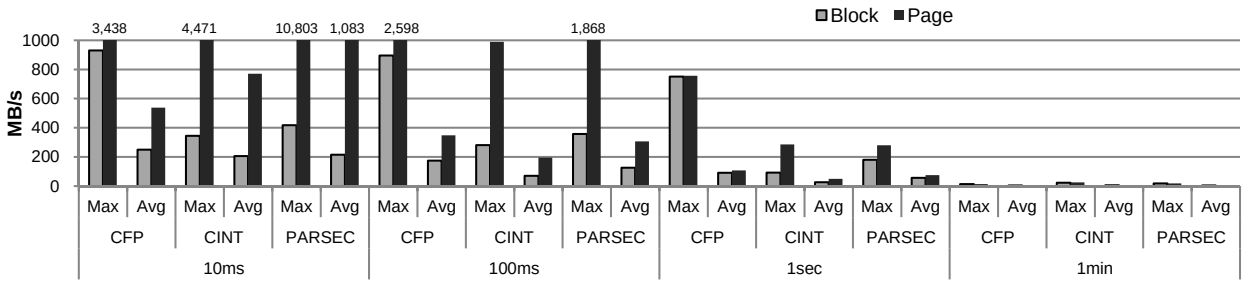

Figure 5. Maximum and average memory bandwidth requirements in MB/s of checkpointing mechanisms for different checkpointing frequencies (in seconds) when using block or page tracking granularities for the SPEC 2006 [30] floating point (CFP), integer (CINT) and PARSEC 2.1 [5] benchmarks.

4. Implementation Details of KIMA

The undo- and redo-log creation mechanisms are tightly inter-dependent as a result of exploiting the synergies between checkpoints. For clarity we will describe them separately in the sections 4.1 and 4.2 respectively and provide details of the design of the KIMA engine in Section 4.3.

4.1 Undo-Log Checkpointing

KIMA's undo-logging uses a mechanism similar to that described in prior work, ReVive [18] and SafetyNet [28]. It uses an extra *checkpoint* bit in the tag arrays for the L1 and L2 caches to identify which blocks were already saved in the current interval. When a block is written in L1, the *checkpoint* bit is checked. If this bit is 1, the block has already been saved in the current undo-log interval and no further KIMA action is needed. If the *checkpoint* bit is 0, the block's data is sent to the undo-log, the *checkpoint* bit is set to 1, and only then the write proceeds to modify the data. When the block is written back to the L2 cache the *checkpoint* bit is written back as well, but for blocks written back and replaced from L2 the *checkpoint* bit information is not preserved (to avoid having to keep these bits for all blocks in memory). Thus, when a block is brought into the L2 cache on a miss, its *checkpoint* bit is assumed to be 0.

Although, the *checkpoint* bit information dramatically reduces the number of writes that reach the KIMA engine, it can still allow multiple copies of the same block to be saved in a given interval ($ULI_{n,m}$); this occurs when a block is saved, replaced form the L2 cache, and then written again. Note that this still allows correct rollback; the undo-log is applied to memory in reverse order and the data value finally restored is one that was saved first. However, this unnecessary saving of already-saved blocks can potentially be a performance and a log-space problem, as we will show in Section 5.

4.2 Redo-Log Checkpoint Creation and Organization

To construct a redo-log checkpoint at the end of a RLI_n, during that RLI_n we need to track the set of modified address for that interval. As we pointed out in Section 3, this set of modified addresses is the same as the set of addresses that were checkpointed by the (many) undo-logs during this redo-log interval. Therefore, whenever a block is sent to an undo-log checkpoint $ULI_{n,m}$, its address is also added to the redo-log meta-data for interval RLI_n. At the end of RLI_n, this meta-data is traversed and the included blocks are copied from DRAM to PCM.

As indicated in Section 3, a significant number of blocks that belong to the redo-log checkpoint RLI_n are also going to be copied by future undo-log checkpoints $ULI_{n+1,m}$. For this reason, we do not start the copying of RLI_n data as soon as that interval ends. Instead, we wait for a period of time in RLI_{n+1}, during which blocks that belong to RLI_n are saved to PCM by undo-log activity. This "forwarding" of undo-log writes is done by checking, for each block that should be saved to the undo-log for $ULI_{n+1,m}$, if 1) the block's address is present in the redo-log meta-data for RLI_n and 2) the corresponding data has not been saved yet. In such a case, the data is saved to PCM, the RLI_n meta-data is updated to point to the saved data, and the $ULI_{n+1,m}$ log is made to point to the same block in PCM.

It is possible (and highly likely) that the sets of modified address of two consecutive redo-log intervals, RLI_n and RLI_{n+1}, are not exactly the same. Therefore, at some point we must stop waiting for undo-logging during RLI_{n+1} to save data that belongs to RLI_n, and start actively copying the remaining data for RLI_n to PCM. We chose to do this at the midpoint of interval RLI_{n+1} (half a second after RLI_n ends). At that point, the KIMA engine traverses the RLI_n meta-data and, for each block that was not already copied (by undo-log activity), reads the block's data from the application's memory and saves it to PCM. Meanwhile, we continue monitoring undo-log activity and saving to PCM any blocks that belong to RLI_n but have not yet been reached by KIMA's redo-log copying activity. This monitoring prevents the application from overwriting any data that should have been saved to RLI_n. Any first modification in RLI_{n+1} will find the *checkpoint* bit in the cache to be zero, and thus send the current (before the modification) value of the block to the undo-logging mechanism. The undo-log mechanism will in turn check the meta-data for the prior redo-log checkpoint, saving the block to PCM and adding its pointer to the redo-log.

A critical aspect of KIMA is the data structure for redo-log meta-data. This data structure is accessed frequently, both during undo- and redo-log checkpoint creation, and needs to be updatable in hardware. We opted for a trie data structure, Figure 6 (only the last 3 levels of the tree are shown for simplicity), which is similar to page-table structures used in today's 64-bit

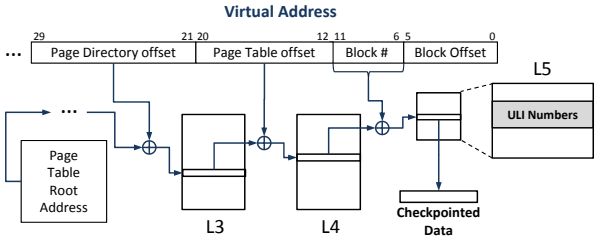

Figure 6. Redo-log checkpoint meta-data trie data structure used by KIMA.

architectures [10]. This structure provides low look-up cost given a block's address (to add addresses and check for overlap during undo-logging), can be traversed in an order sorted by address (this facilitates consolidation of checkpoints), and there are existing hardware mechanisms for accessing its elements in modern processors[3]. A typical page table consists of four levels where the last level (L4) points to the physical address of a page in memory. In KIMA, though, we checkpoint memory at the granularity of blocks, so we add an additional level (L5) that contains 64 pointers, for the respective memory blocks of a memory page, which point to the checkpointed data in PCM.

An important additional benefit of updating the trie structure for RLI_{n+1} when saving data for ($ULI_{n+1,m}$) is that we can perform a secondary filtering to prevent redundant undo-log entries. As explained in Section 4.1, such redundant entries are not entirely eliminated by adding *checkpoint* bits to on-chip caches. To do this secondary filtering, each entry in the redo-log trie also keeps the number of the last undo-long interval that checkpointed that entry. If the check finds the number of the current undo-log interval in the redo-log entry, the block has already been saved and does not need to be saved again (neither in the undo-log nor in any redo-log). Since there is a limited number of undo-log intervals within a redo-log interval (up to 100, for 10ms undo-log intervals), this only requires 7 bits in the redo-log entry, and we use the block offset bits and the unused bits of the virtual address [4] (which are not needed) to store the ULI number.

The PCM memory is managed as a free list of blocks maintained by the OS. When copying a block to PCM, KIMA gets the PCM location from one end of the free list. When redo-logs are consolidated, freed blocks are returned to the other end of the free list. This "rotation" of blocks through the free list helps prevent excessive writes to any one block of PCM memory.

Finally, the redo-log meta-data are stored in DRAM and the internal KIMA engine caches them during the time period when they are updated. Only after all the blocks of the redo-log checkpoint have been copied to PCM, KIMA starts copying the corresponding meta-data to PCM, starting from the leaves of the trie structure. When the root of the meta-data and the corresponding pointer to it have been stored to PCM, the checkpoint is complete and becomes part of the redo-log recovery state.

[3] Other data structures, such as hash-tables, could be potentially used but the trie structure proved to satisfy KIMA's functionality requirements

[4] In x86 64-bit architectures the physical address-space is limited to 48 bits [10] and additional bits can be used to represent the ULI number if necessary.

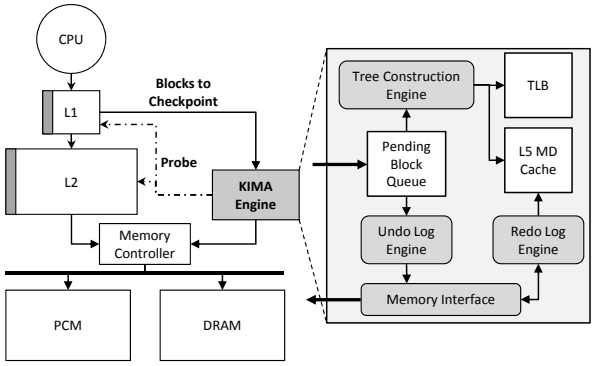

Figure 7. Architecture of the KIMA hardware engine.

4.3 KIMA Engine Description

The KIMA hardware engine (Figure 7) is responsible for constructing both the undo-log and the redo-log checkpoints. It is a structure separate from the cores of the CMP, and is positioned close to the on-chip memory controller for minimizing the latency to memory. The number of KIMA engines on chip will depend on the number of cores, and the checkpointing requirements of future systems. The engine receives the blocks to be checkpointed from the L1 caches of the cores[5]. In order to differentiate between the processes/threads running concurrently on the CMP, along with the data to be checkpointed, the L1 also sends the number of the core which modified the block. Based on the core number the KIMA engine selects the appropriate undo-log and redo-log checkpoints to insert the data. The OS is responsible for programming and managing the KIMA engine. When a new process/thread is scheduled in one of the cores, the OS updates the KIMA engine registers with the following values: 1) the pointers of the undo-log data and meta-data, 2) the pointers to the roots of the redo-log meta-data for the current and the previous redo-log intervals (RLI_{n+1} and RLI_n), and 3) the core where it will be running. Threads of the same processes, which typically share the same address-space, will have the same set of pointers to undo- and redo-log meta-data.

Internally the KIMA engine consists of a pending queue of blocks to be processed, an undo-log engine (ULE), a tree construction engine (TCE), a redo-log construction engine (RLCE) and a memory interface. Each core sends the blocks to be saved in the KIMA engine and they are inserted in the pending queue. Every block in the pending queue is first processed by the *TCE*, which updates the meta-data for the current redo-log (RLI_{n+1}), performs secondary filtering of undo-log writes, and uses the RLI_n meta-data to decide whether the block will be saved to DRAM or PCM. Once the block is processed by the *TCE* it is forwarded to the undo-log en-

[5] In our architecture we are assuming a snooping cache coherency protocol. In the case of a directory based protocol, the directory would be responsible for identifying the blocks to be checkpointed, similar to ReVive [18]

gine, which writes the block to either DRAM or PCM and updates the undo-log meta-data, by issuing requests to the memory interface. During the first half of RLI_{n+1} the checkpoint for RLI_n is updated with the common blocks in $ULI_{n+1,m}$ which are saved to PCM. When we cross the half-second point in RLI_{n+1} the redo-log construction engine starts walking the redo-log meta-data for RLI_n and copying from DRAM to PCM the blocks that have not been checkpointed yet, by sending requests to the memory interface. It is possible that data to be checkpointed still reside in the caches, for this reason the KIMA engine behaves just like a core, using the existing cache coherence to get the most recent data either from on-chip caches or, if the data is not on-chip, from memory.

The KIMA engine updates the meta-data of RLI_n and RLI_{n+1} frequently. To reduce memory bus activity when accessing this meta-data the KIMA engine has a translation look-aside buffer (TLB), where it caches only the first four levels of the meta-data trie structure, and also a small cache for storing the ULI numbers contained in the last level (L5) nodes of the trie (L5 MD Cache). The intermediate nodes of the trie are being accessed on every look-up of the trie, and caching them separately eliminates any misses that the co-location with the L5 node data would generate.

5. Evaluation

We quantitatively evaluate the hardware cost, the performance overhead and the memory requirements of KIMA.

5.1 Hardware Configuration

In our evaluation, we use SESC [22], 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. PCM memory shares the same bus as DRAM and has an average read latency of 150ns and write latency of 450ns (450 and 1350 cycles respectively) [27]. The KIMA engine we simulate has a 64 entry pending block queue, a 256 entry fully associative trie TLB, a 64KB 16-way associative single-ported L5 MD cache, and the memory interface has a 32 entry read queue and a 128 entry write queue. In total, the KIMA engine requires $\sim$82KB of on-chip state, which is lower than 256KB in SafetyNet [28]. Because this state is kept in area-optimized, single ported arrays, its area is 40% smaller than the area of a single core's L1 cache (estimated using CACTI 5.3 [31])

5.2 Evaluation Methodology

In our evaluation we compare KIMA against a number of possible alternative hardware and software checkpointing techniques. Some of these alternatives correspond to state-of-the-art implementations and some represent partial implementations of KIMA-like enhancements for the purpose of isolating the contributions of each enhancement. All these mechanisms use hardware undo-logging similar to ReVive [18] and SafetyNet [28], and they differ in how they create redo-logs:

Software Page Based (SW-Page): This implementation uses a software thread for redo-logging in parallel with the program execution. It leverages existing dirty-bit information embedded in page-tables to identify modified pages of a given RLI and then copies those pages to PCM. To prevent the application from modifying data that have not been checkpointed yet, it write-protects such pages; when the application attempts a write, an exception handler immediately copies the page to PCM and resumes the application's execution.

Software Page Based No Cache Allocate (SW-Page-NCA): This implementation is similar to the previous one with the only exception that data to be read or written for creating the redo-log checkpoint are not allocated in the caches, but are being copied with the help of streaming buffers. This improves cache performance because redo-log data has little or no temporal locality.

Software Block Based (SW-Block): This approach is practically a software implementation of the *decoupled engine* (which will be described), but uses a software thread instead of a hardware redo-logging engine to 1) read undo-log meta-data to update the redo-log trie structure and construct the list of blocks to copy and 2) checkpoint these blocks to PCM. Just like the page-based schemes that were described previously, it uses memory protection to prevent the corruption of the data that is yet to be copied, and just like SW-Page-NCA, cache pollution with low-locality data is avoided by using streaming buffers and bypassing the caches for such data.

Hare and Undo-Log (HARE+Ulog): This approach leverages an existing hardware mechanism which establishes redo-log checkpoints, HARE [8]. In our evaluation we use only the checkpointing engine of HARE as it was originally proposed, and we disable its checkpoint merging functionality.

Decoupled Engine (Dec-Eng): The *decoupled engine* is similar in structure to the KIMA engine, but the undo- and redo-logging are performed independently. The decoupled engine saves undo-log data and meta-data into logs without updating the redo-logging trie structure. The redo-logging engine reads undo-log meta-data and build its trie, then copies data to the redo-log. This decoupled approach leverages the first observation we make (undo- and redo-logging save data for the same addresses) but cannot benefit from the second observation (undo-log data copying can help redo-log data copying). It also cannot benefit from the trie-based secondary filtering in undo-logs. Because the trie structure needs no data pointers prior to active redo-log copying, the decoupled engine does not need a cache for L5 nodes. Instead, the trie TLB entries are used as 64 bit-masks to mark modified blocks in that page. This reduces the cost of the engine – its on-chip state is only 22KB.

Our evaluation uses 27 of the 29 SPEC 2006 [30] benchmarks, shown in Figure 8. The only benchmarks omitted are tonto and perl because of incompatibilities with our simulator infrastructure. We simulate SPEC benchmarks using reference

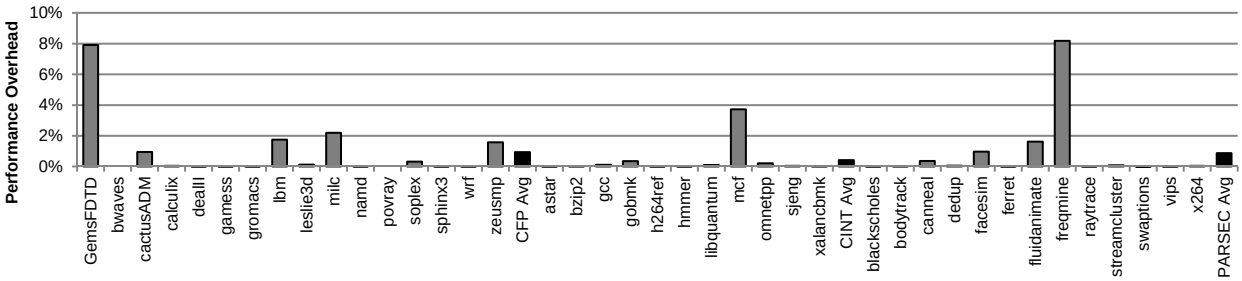

Figure 8. Performance overhead of KIMA for the SPECFP 2006 (CFP), SPECINT (CINT), and PARSEC benchmarks.

inputs, by fast-forwarding 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 evaluate KIMA with all 13 multi-threaded benchmarks from PARSEC 2.1 [5], using native inputs and four threads. The exception is dedup where we use the simlarge input because the native input exceeds the addresses-space of the 32-bit MIPS-Linux simulated machine in SESC. We fast-forward PARSEC applications to the beginning of the parallel execution, fast-forward an additional 21 billion instructions to warm-up the memory tracking mechanisms of all undo- and redo-logging techniques, then simulate 20 billion instructions in detail. In the evaluation of software-based checkpointing techniques for multi-threaded workloads, checkpointing threads always have higher priority than application threads so one of the application threads is suspended when the checkpointing thread is active in case of no "free" cores.

5.3 Performance Results

Performance Overhead of KIMA. Figure 8 presents the performance overhead of KIMA for all simulated applications, along with the averages for SPECFP, SPECINT and PARSEC benchmarks. The undo-log checkpointing period is 10ms and the redo-log period is 1 second. Overall, KIMA has average performance overheads of $\sim$1%, and the maximum overhead is 8% (in GemsFDTD and freqmine). The benchmarks where KIMA has the highest overheads are applications which are already memory intensive, memory bandwidth constrained and are creating the largest checkpoints across all applications we have evaluated. KIMA's overheads are still low in these applications because, KIMA efficiently reduces the memory bandwidth needs of redo-log checkpoints when compared to other checkpointing techniques.

Comparison of KIMA with other techniques. Figure 9 compares performance overheads of the five alternative techniques described in Section 5.2. All schemes create undo-log checkpoints every 10ms and redo-log checkpoints every 1 second. In the Figure we present the worst performing benchmarks for KIMA, along with averages for all benchmarks in the SPECFP, SPECINT and PARSEC suites. KIMA outperforms all alternative techniques across all benchmarks, reducing overheads by one order of magnitude in some cases. The only exception is freqmine which does not benefit from KIMA's memory bandwidth optimizations. Even so, KIMA's overhead in freqmine is relatively low (8%) and similar to that of the best-performing technique.

To understand why KIMA has an advantage over other techniques, we use the comparison from Figure 9 to identify the causes of performance overhead. Software techniques compete with the application for space in the shared caches. This competition for cache space can be quantified by comparing the SW-Page and SW-Page-NCA techniques. Examples of applications where cache space contention has a significant impact are GemsFDTD and mcf, whose overhead reduces by 10%. However other applications and the averages indicate the elimination of cache contention is not the main reason for KIMA's good performance.

Another source of overhead is contention for memory bus bandwidth. The first step to reduce bandwidth consumed for checkpointing is to use finer memory tracking granularities and eliminate unnecessarily copied data. SW-Block tries to do this in software, providing a significant benefit in milc and mcf – in these applications page-based redo-logging copies twice as much data as block-based SW-Block does. However SW-Block achieves its finer granularity at a cost of doing much more work to construct its set of data to be copied, which results in significant new overheads in benchmarks like lbm, facesim[6], etc.

After eliminating cache contention and reducing bus bandwidth contention by using finer tracking granularities, the next step to improve performance of redo-logging is to use specialized hardware. In our experimentation we observed that more than 50% of the overhead of software implementations comes from pausing the application's execution in order to serve a page-fault caused when the application tries to modify data which have not been checkpointed yet. Hardware techniques dramatically reduce these pauses by creating checkpoints faster.

Note that Dec-Eng and HARE+Ulog have similar overheads because both need to copy the same amount of data in order to establish a redo-log checkpoint. The marginally higher overheads of HARE+Ulog compared to Dec-Eng (with the exception of GemsFDTD where HARE+Ulog is 5% slower) are caused by the caching of HARE's memory modification tracking meta-data in the L2 cache and the additional cost of sorting the collision list [8]. KIMA outperforms both Dec-Eng and

[6]This cost cannot be hidden for the case of the PARSEC benchmarks because the checkpointing thread preempts the application's threads, while for the SPEC benchmarks the checkpointing thread runs on one of the idle cores

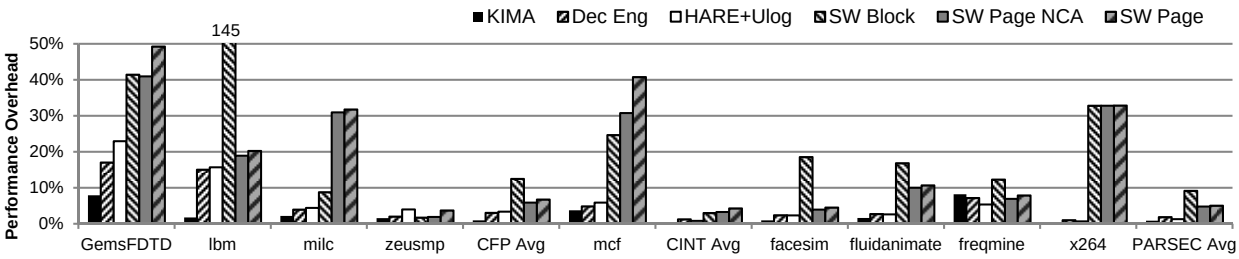

Figure 9. Worst case and average performance overhead numbers of KIMA when compared to other checkpointing approaches for the SPECFP (CFP), SPECINT (CINT) and PARSEC benchmarks.

HARE+Ulog because it further reduces the bandwidth requirements by 1) eliminating the reads of undo-log meta-data that Dec-Eng does, 2) removing duplicate entries in the undo-logs via secondary trie-based filtering and 3) avoiding many reads and writes in the redo-log copying by leveraging copying activity of undo-logging. As shown in Figure 9, these benefits can be very significant especially in the applications that suffer the highest overheads in Dec-Eng and HARE+Ulog.

5.4 Sensitivity Analysis

Another way to reduce the memory bandwidth requirements of a combined undo-log and redo-log checkpoint mechanism is to decrease checkpointing frequency of undo-logs. The intuition behind this approach is that blocks that are checkpointed multiple times across consecutive undo-log checkpoints are going to be checkpointed only once and the memory bandwidth requirements are going to decrease. In Figure 10 we present the highest overhead benchmarks along the with the averages for KIMA and Dec-Eng for undo-log checkpointing frequencies of 10ms and 100ms, keeping the redo-log frequency unchanged (at 1 second). KIMA benefits from the reduced checkpointing frequency, although only marginally. Contrary to our expectations, the overhead of Dec-Eng actually increases when undo-log checkpoints are created less often.

To gain better insight into this phenomenon, Figure 11 shows the breakdown of memory accesses into: accesses generated by the application (App), necessary undo-log writes (UL-Nec), unnecessary undo-log writes (UL-Unnec), redo-log construction reads and writes (RL), and engine accesses (EA) (such as the reads of the undo-log meta-data for Dec-Eng and the misses from the engine's caching structures in both Dec-Eng and KIMA). All numbers are normalized to application accesses (App) of the slowest configuration (Dec-Eng with 100ms undo-logging).

We observe that, by decreasing the undo-log frequency, the number of unnecessary checkpointed undo-log blocks increases for Dec-Eng. The reason for this is that long undo-log checkpointing intervals increase the probability of a block getting replaced for the L2 cache after it has been checkpointed. This removes the *checkpoint* bit information, resulting in the same block being checkpointed again when it is written again in the same undo-log interval. KIMA does not suffer from this behavior because of its secondary trie-based filtering that prevents duplicates from being saved to the undo-log.

Additional performance benefit of KIMA over Dec-Eng (and HARE+Ulog) comes from overlapping undo-log copying

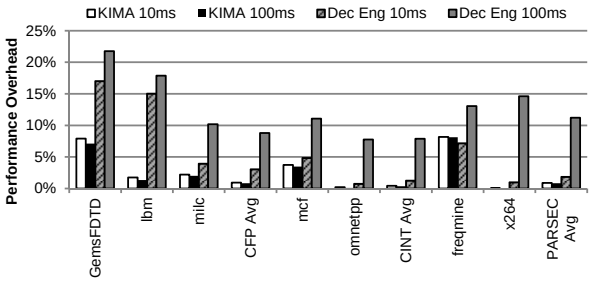

Figure 10. Worst case and average performance overhead numbers of KIMA compared to the decoupled checkpointing engine for undo-log checkpointing frequencies of 10ms and 100ms.

for redo-log checkpoints. This approach eliminates almost all redo-log copying activity in lbm, and reduces such activity by 50% percent in GemsFDTD. Overall, this synergistic copying eliminates 50% of redo-log copying activity on average across all applications.

We have also performed experiments (Figure 12) where we vary the undo-log (0.01 and 0.1 sec) and redo-log (0.5, 1 and 2 sec) checkpointing frequencies. As expected, higher redo-log checkpointing frequencies (0.5sec) increase the overhead of KIMA to 2% on average for CFP. The increased overhead can be attributed to the additional copied memory for creating twice as many checkpoints, and also to the reduced percentage of blocks forwarded between undo- and redo-log checkpoints[7]. This observation explains why the overhead increases for GemsFDTD and lbm when we increase the undo-log interval from 0.01sec to 0.1sec and maintain the redo-log frequency at 0.5sec: shorter redo-log intervals do not match the application's memory modification period, so there are fewer common blocks between consecutive redo-log intervals, resulting in reduced synergistic copying. For lower redo-log checkpointing frequencies (2sec) the overhead decreases, as expected, because we create fewer checkpoints, and also because close to 100% of redo-log blocks come from the undo-logs (it was only 50% for GemsFDTD for redo-log frequency of 1sec).

The only benchmark where KIMA offers no performance improvement over the other techniques is freqmine (Figure 9). Freqmine has a memory access pattern such that: 1) no unnecessary undo-log entries are created (no benefit from secondary filtering), 2) there is limited number of synergistic copying op-

[7] It is enabled only during the first half of the redo-log interval

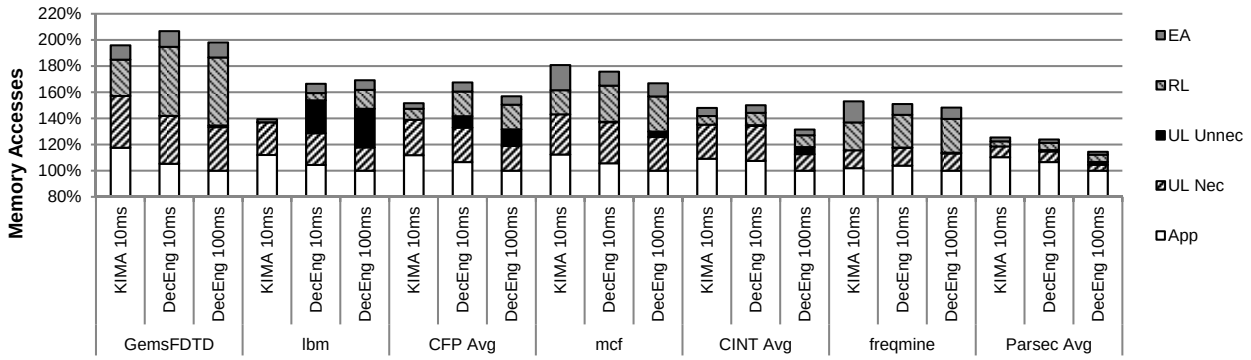

Figure 11. Breakdown of the type of memory accesses of the KIMA and the decoupled memory engine for undo-log checkpointing frequencies of 10ms and 100ms.

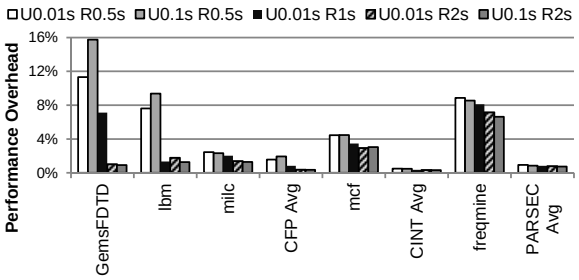

Figure 12. Worst case and average performance overhead numbers of KIMA for different undo-log (U) and redo-log (R) checkpointing frequencies.

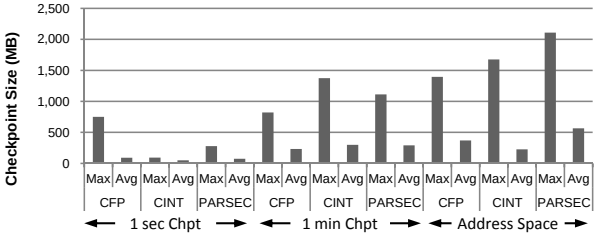

Figure 13. Maximum and average memory requirements in MB of the one second, one minute checkpoints and the address-space of the application.

portunities when creating redo-log checkpoints every 1sec, but can improve as we decrease the checkpointing frequency (Figure 12), and 3) KIMA caches exhibit high miss-rates. As part of our future work we plan to continue studying the performance benefits of synergistic undo/redo-log creation and further reduce the associated performance cost.

5.5 PCM Memory Requirements and Performance/Power Considerations

To estimate the PCM space requirements of checkpointing we profiled the SPEC and PARSEC applications using PIN [13] (we used the reference and native inputs respectively and the applications ran to completion) and estimated the average redo-log checkpoint size at 1-second and 1-minute frequencies. The benchmarks we used in our evaluation do not run long enough to estimate the size of 1-hour checkpoints, so we assume that a 1-hour checkpoints is as large as the allocated address space of the application. The results are shown in Figure 13. We find that 4GB of PCM would be sufficient on average to store all redo-log checkpoints (1s, 1min, 1h). In KIMA, we copy the old one-hour checkpoints to other storage media (flash or hard disks). Our experiments indicate that this requires a maximum bandwidth of 35MB/sec, which can be provided by existing storage. Note that 4GB PCM requirement is not fundamental – if needed we could adopt more aggressive consolidation, based on the policy dictated by the recovery mechanism. We also estimated the lifetime of a 4GB PCM with KIMA – 17.7 years in the worst case. KIMA writes to PCM only one checkpoint per second, consolidations update only meta-data, and writes can easily be distributed over PCM to avoid waring out a single block.

PCM is not a standardized memory technology yet and the expected write latency may increase in final products. For this reason, we also conducted experiments where we doubled the average write latency of PCM to 900ns or 2700 cycles for our architecture. In these experiments, the performance overhead of KIMA remains the same, and there is a minor increase in the checkpoint creation time. The 128-entry write queue of the KIMA engine, which is 50% to 60% occupied in our original experiments, has 90% occupancy in increased-latency experiments. With this in mind, we except that further increases in the PCM latency will require larger write queues to avoid noticeable performance impact.

Finally, we estimate the power overhead of our technique to the system, using CACTI 5.3 [31] to determine the dynamic power consumed by the caches/queues of the KIMA engine and the DRAM memory of the system. PCM is expected to have higher write power requirements than DRAM and we used the power estimates from Lee *et. al.* [12]. These are expected to be the dominant sources of additional power consumption caused by KIMA. Our results show that the majority of the power overhead is caused by the additional memory accesses for creating the checkpoints, and that the power requirements of the system increase by 1-2 Watts on average.

6. Conclusions

Reliability is going to become an increasing problem in the future with recovery mechanisms playing a critical role in the

availability of future systems. The two main approaches to checkpointing each have important limitations – redo-logs can recover from catastrophic faults but have poor recovery latencies for the most common errors, while undo-logs offer quick recovery from common errors but cannot recover from catastrophic faults.

In this paper we present KIMA, a mechanism that synergistically combines the two approaches. It creates undo-log and redo-log checkpoints together, using their inherent relationships to minimize the performance overhead, bandwidth consumption, and memory space needed for this purpose. Our experiments indicate that KIMA incurs minimal overheads ($\sim 1\%$) on average and 8% worst-case across a variety of single-threaded and multi-threaded benchmarks, while supporting efficient recovery from catastrophic and non-catastrophic errors that have a wide range of detection latencies (from a few milliseconds to one hour).

References

[1] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. *2005 Intl. Conf. on Dependable Systems and Networks (DSN'05)*, pages 12–21, 2005.

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

[3] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[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] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *ACM SIGOPS 22nd Symp. on Operating Sys. Principles*, page 133, 2009.

[7] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems. In *Conf. on High Perf. Computing Networking, Storage and Analysis - SC '09*, page 1, 2009.

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

[9] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *15th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 385–396, 2010.

[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] International Technology Roadmap for Semiconductors. Process intergration, devices & structures. 2007.

[12] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *36th Intl. Symp. on Comp. Arch.*, page 2, 2009.

[13] 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.

[14] A. Moody, G. Bronevetsky, and K. Mohror. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. *Proc. of the 2010 IEEE/ACM Conf. on Supercomputing*, 2010.

[15] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *Proc. of 12th Intl. Symp. on High-Perf. Comp. Arch.*, 2006.

[16] R. a. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *24th IEEE Conf. on Mass Storage Systems and Technologies (MSST 2007)*, number Msst, pages 30–46, 2007.

[17] 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.

[18] 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.

[19] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *36th Intl. Symp. on Comp. Arch.*, volume 37, page 24, 2009.

[20] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based Fault Screening. *Proc. of 13th Intl. Symp. on High Perf. Comp. Arch.*, 2007.

[21] D. Reed. High-End Computing: The Challenge of Scale. *Director's Colloquium*, 2004.

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

[23] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *42nd IEEE/ACM Intl. Symp. on Microarchitecture*, page 122, 2009.

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

[25] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conf. Series*, 78, 2007.

[26] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild. In *11th Intl. Joint Conf. on Measurement and Modeling of Computer Systems*, page 193, 2009.

[27] N. H. Seong, D. H. Woo, and H.-h. S. Lee. Security Refresh: Prevent MaliciousWear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping. In *37th Intl. Symp. on Comp. Arch.*, page 383, 2010.

[28] 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.

[29] 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.

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

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

[32] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. In *IEEE Trans. on Dependable and Secure Computing*, volume 3, pages 188–201, 2006.

[33] 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.

[34] H. Yu, R. Sahoo, C. Howson, G. Almasi, J. Castanos, M. Gupta, J. Moreira, J. Parker, T. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. Gropp. High Performance File I/O for The Blue Gene/L Supercomputer. In *The 12th Intl. Symp. on High-Perf. Comp. Arch., 2006.*, pages 190–199, 2006.