Adaptive Incremental Checkpointing for Massively Parallel Systems

Saurabh Agarwal, Rahul Garg, Meeta S. Gupta
IBM India Research Labs
Block -1 IIT Hauz Khas, New Delhi, India
(saurabh.agarwal, grahul, meetasha)@in.ibm.com

Jose E. Moreira
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
moreira@us.ibm.com

ABSTRACT

Given the scale of massively parallel systems, occurrence of faults is no longer an exception but a regular event. Periodic checkpointing is becoming increasingly important in these systems. However, huge memory footprints of parallel applications place severe limitations on scalability of normal checkpointing techniques. Incremental checkpointing is a well researched technique that addresses scalability concerns, but most of the implementations require paging support from hardware and the underlying operating system, which may not be always available. In this paper, we propose a software based *adaptive* incremental checkpoint technique which uses a *secure hash* function to uniquely identify changed blocks in memory. Our algorithm is the first *self-optimizing* algorithm that dynamically computes the optimal block boundaries, based on the history of changed blocks. This provides better opportunities for minimizing checkpoint file size. Since the hash is computed in software, we do not need any system support for this. We have implemented and tested this mechanism on the BlueGene/L system. Our results on several well-known benchmarks are encouraging, both in terms of reduction in average checkpoint file size and adaptivity towards application's memory access patterns.

Categories and Subject Descriptors: D.4.5 Checkpoint and Restart, Fault-Tolerance
General Terms: Algorithms, Performance, Reliability, Experimentation
Keywords: Fault-Tolerance, Incremental Checkpoint, Large Scale Systems, Probabilistic Checkpoint

1. INTRODUCTION

High performance computing systems are now being increasingly built off the commodity hardware with high-speed interconnects, to minimize cost. As more of such systems are being deployed in practice, issues of fault-tolerance and self-healing are becoming tremendously important. While larger systems are being developed for better performance, it exponentially increases the total number of failure points in the system. For instance, if mean-time-between-failure (MTBF) of an individual node is ten years, the MTBF of a sixty four thousand node system would be only 1.37 hrs (assuming independently and exponentially distributed time before failure). Moreover, these machines are increasingly being used by the scientific community to solve computationally intensive problems which typically run for days or even months. It is therefore absolutely essential that these long-running applications are able to tolerate hardware and software failures and avoid re-computations from scratch. While hardware fault-tolerance can be achieved by deploying redundant hardware components and re-allocating alternate hardware resources to the applications at run-time, this approach is not cost-effective. The age-old checkpoint/restart mechanism[1] still seems most attractive due to its simplicity and low-cost, except that the algorithms need to be adaptive and scalable to the size of current systems. The high failure rate of these systems puts additional pressure on checkpoint mechanisms. Checkpoints should now be taken more frequently [1, 2] relative to the failure rate of the system which, in turn, directly impacts the application running time and disk-storage requirements. One of the ways to reduce the checkpoint file size is incremental checkpointing technique, as proposed and implemented earlier by several researchers [3, 4, 5, 6, 7], details of which are discussed in section 2.

In this paper, we present our experiences in designing a novel, incremental checkpoint algorithm for the BlueGene/L machine. Ours is a software-only approach, which attempts to dynamically identify *near-exact* [2] changed blocks of memory so that only minimal data is stored on the disk. The application memory is initially split into blocks of equal sizes. At every checkpoint, the memory partitioning is refined, based on the past access patterns of the application. Thus, with every step, the algorithm attempts to reduce the size of changed blocks to the actual amount changed, while increase the size of un-changed blocks. Over a period of time, the algorithm aims to identify near-optimal sizes of the changed blocks, thereby reducing the checkpoint file size to just the bare minimum required. To the best of our knowledge, this is the first self-optimizing checkpointing implementation that tracks application's memory requirements

ICS'04, June 26–July 1, 2004, Saint-Malo, France.

[1]Periodically saving the process state to stable storage, and in the event of a failure, restart from the most recent checkpoint.
[2]By *near-exact*, we mean a block as close to the exact changed bytes in size as possible (usually greater), limited by the adaptivity of the algorithm on a particular dataset.

and data access patterns in order to decide how much and exactly what to save.

Our experimental results obtained on a small 8 node BlueGene/L partition indicate that incremental checkpointing is a viable technique for providing fault-tolerance to long-running scientific applications. The proposed adaptive incremental checkpointing seems to hold better potential (w.r.t reduced average checkpoint file size, reduced checkpointing time and automatic block-size tuning) than previous incremental checkpointing approaches, although more experimentation on larger platforms (which will be soon available to us for testing) and further fine-tuning is needed to draw very strong conclusions.

The rest of the paper is organized as follows. Section 2 discusses the previous work on checkpointing mechanisms in general and the incremental checkpointing technique in particular. Section 3 provides a quick overview of the Application Programmer's Interface (API) and some notable features of our checkpoint library. In Section 4, we discuss hash-based incremental checkpointing technique (using fixed blocks) in greater details, to further motivate the need for an adaptive mechanism. The algorithmic and implementation details of our proposed adaptive incremental checkpointing technique are then presented in Section 5. Description of our experimental setup and a detailed performance analysis of the the results obtained for various schemes is presented in Section 6. Finally, Section 7 draws conclusions from this work and Section 8 presents our directions into the future.

2. RELATED WORK

Checkpointing with rollback-recovery is a well known technique for fault-tolerance in parallel and distributed computing space. There has been much work in the field of distributed checkpointing and rollback recovery. Refer to [8] for an excellent survey. However, none of these systems convincingly demonstrate the scalability that could match the scale of tomorrow's largest supercomputers like BlueGene/L. To the best of our knowledge, the largest known checkpointing system demonstrated is CLIP [9], which ran experiments on a 128 processors Intel paragon machine with 32MB of memory on each node. Many other implementations are available at various levels of abstraction: application-level [9, 10, 7, 11], compiler supported [12, 13], runtime-library supported [14, 15, 16, 17], operating system supported [18, 19, 20] and hardware-assisted [21, 22]. Clearly, each of these abstractions have their respective advantages and disadvantages, and one has to make trade-offs with respect to factors like user-transparency, portability, flexibility and granularity of the checkpointing desired. An excellent description of complete design space and associated trade-offs is given in a recent study by Sancho et. al [23]. In our work however, the focus is on the performance and scalability of the checkpointing technique, and in this context we believe (and as has also been noted by [23]) that incremental checkpointing approach holds most promise.

Incremental checkpoint techniques can broadly be classified into page-based or hash-based techniques. Page-based techniques require memory-protection hardware and support from the operating system (OS) to be able to identify dirty pages. Hash based techniques, on the other hand, use a hash function to compare and identify the changed blocks of memory. Most incremental techniques implemented in practice [3, 4, 5, 6, 7] have used OS supported page-protection mechanism. However, there are two fundamental limitations of this approach. Firstly, such a scheme requires memory-protection support from the underlying hardware along with support from the OS to be able to handle page-fault exceptions. While such a feature might almost sound *granted* in most modern hardware and OS, many designers opt not to enable such features for reasons of simplicity and performance (e.g., BlueGene/L). Secondly, a more serious limitation of the page-based approach is that there is no easy way to identify the exact changed bytes in a page. Even if a single byte is changed in a page, the entire page needs to be stored in the checkpoint. Although this is a well known limitation [24], detecting *exact* changed words of memory is way too time consuming [13, 24], which motivates the need for another approach that could identify *near-exact* changed blocks, without compromising much on the additional time it may take to do so.

One way to do so is to reduce the page size of the system, so it would be possible to identify changed words at a smaller granularity. Although this can be done by simply re-compiling the kernel, this is un-attractive because reducing the system page size arbitrarily has serious implications on the overall system performance. For one, small page size results into a large page table, which leads to more TLB misses, which in turn increases the runtime overhead of the application. Besides, for the sake of argument, even if we decided to ignore overall system performance, determining the optimal system page size is not so easy, as it will vary from one application to another. In summary, we are of the opinion that tampering with the page size of the memory management subsystem for seeking performance benefits in incremental checkpointing is not a good idea.

This brings us to discuss the second approach towards incremental checkpointing, the hash-based technique. To the best of our knowledge, there is very limited work on hash based techniques, and most of this work is only at the proof-of-concept stages. The initial work on probabilistic checkpointing [24] was seriously challenged in [25], where it was shown that hash functions used in [24] (checksum and CRC32) will, with a non-negligible probability, lead to collisions thereby making the approach incorrect. It was later shown in [26] that using secure hash functions (like MD5, SHA1 etc) it was possible to implement reliable incremental checkpointing technique. However, none of these approaches utilize the full potential of the hashing technique. We will further discuss on these approaches in section 4, when we discuss our own implementation of this technique.

3. CHECKPOINT LIBRARY API AND FEATURES

We have implemented an application-level checkpoint library for the BlueGene/L system. Currently, checkpoint is user-initiated, while the restart is transparent[3]. The application writer is expected to register the application with the checkpoint library through an explicit call to *BGLCheckpointInit* in the initialization section of the code. This function initializes the checkpoint data structures and automatically decides weather or not a restart is needed, based on the environment variable settings (which, in turn, will be automatically set by the job scheduling sub-system). In or-

[3]We plan to extend this to completely transparent and asynchronous design in future.

der to take a checkpoint, the application writer is expected to explicitly call the function *BGLCheckpoint()* whenever a checkpoint is required. Additionally, application writer must also ensure that call is made at a point when there will be no in-flight messages in the network (typically after an *MPIBarrier*). The call automatically saves complete application's state including its I/O and signal states[4] into the checkpoint file. A modified ELF file format is used to save the checkpoint file. The extensibility and flexibility of the ELF format was very useful in defining additional sections required for incremental checkpoint. Some of the most useful features provided are:

Signal Handling: Support has been added to the checkpoint library to handle signals at any point (before, during and after a checkpoint). If application installs a signal handler (through *signal()* call) in the code, checkpoint library intercepts it, and installs its own handler instead. Application handlers cannot be called while a checkpoint is in progress, as they may clobber the application's state. If an urgent signal arrives while application is taking a checkpoint, the checkpoint is aborted, and the signal is delivered to the application. Other signals are kept pending, and are delivered to the application once the checkpoint is complete. Signals which are un-trappable (SIGKILL, SIGSTOP) are never handled by the library, and the system takes the default action against them.

Callback functions: An application can register functions to be called at various stages of checkpoint. Functions registered through the *AtCheckpoint(), AtContinue()* and *AtRestart()* APIs are respectively called before the checkpoint begins, after the checkpoint completes and after the state is restored in a restart. These functions can be used to quiesce the network and carry out other bookkeeping operations if needed.

Memory Exclusion: *BGLCheckpointExcludeRegion()* primitive allows application writers to exclude specified region of the memory from the checkpoint file. This can be used to further reduce the size of the checkpoint file, but this call places assumptions on the application writer's ability to be able to discern what can and what cannot be excluded. The function is also used internally by the library for protecting its internal data structures.

Checkpoint file verification: Checkpoints are critical for fault-tolerance in the BlueGene/L system, and hence it is essential to ensure that all checkpoints are correctly taken and that they have not been corrupted when needed for restart. In order to detect file corruption, every ELF section written to the checkpoint file is protected with an independent hash, which is stored in the respective section headers. Finally, the ELF header and the section header table are also protected. This ensures that we not only detect that a checkpoint file is corrupted, but also localize the part(s) of the file that has been corrupted. MD5 based secure hash function is used in lieu of simpler CRC to ensure better error detection abilities, in case of file corruption.

4. HASH-BASED INCREMENTAL CHECKPOINT

Noting the limitations in the page-based approach (as discussed in section 2), an alternative approach was proposed in [24], that would make use of a hash function to identify the changed blocks of memory. In a hash-based checkpoint, a hash function $H()$ maps a block B of memory to a unique value $H(B)$, called the hash value of the block. After taking a checkpoint, the hash of each of the memory block is computed and stored in a *hash table*. At the time of taking the next checkpoint, the hash of each block of the memory is recomputed and compared against the value previously stored in the hash table. If the two hash values differ, the block is declared *changed*, and is stored as a part of the checkpoint file. Otherwise, the block is *optimistically assumed* to be unchanged, and is not stored. There are two important aspects to be considered in this approach, which require careful investigation. First, the choice of a hash function and second, the granularity of the blocks. The choice of the hash function impacts two factors: correctness and performance, the first being obviously more serious.

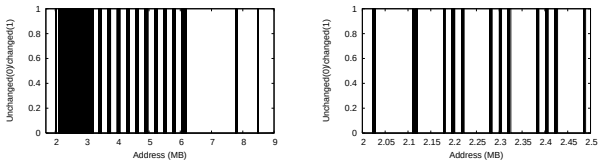
Choice of suitable hash function: The hash function is chosen such that the range of its values is significantly smaller than its domain. This makes the space needed to store the hash value significantly smaller than the size of the memory blocks. If h is the space needed to store the hash values, b is the block size, and a is the size of other entries in the hash table (block address, size etc.), then the hash table overhead is $(h+a)/b$ fraction of the application memory. If h and b are suitably chosen, hash table will take only a small fraction of the application memory. If h is smaller than b then many distinct memory blocks can have the same hash value. In fact, from the pigeon hole principle it turns out that at least 2^{b-h} distinct memory blocks will have the same hash value. This raises serious issues about the correctness of the checkpoint. A memory block may be changed from one checkpoint to the next checkpoint in such a way that its hash value remains the same. Such an event is called a *hash collision* (also called aliasing in [25]). In such an event, the colliding block will be identified as unchanged and will not be stored in the checkpoint, leading to an incorrect checkpoint. However, if the hash function is chosen carefully, the probability of such an even can be made extremely small.

A hash function is called uniform if for any hash value v, the probability that a randomly chosen block B will have a hash $H(B)$ equal to v is 2^{-h} (where h is measured in bits). Note that this decays exponentially with h. Thus, if h is sufficiently large and then the probability of collision can be made extremely small.

It can be seen that functions such as checksum, blockwise XOR and CRC are uniform hash functions. However, it has been reported that, in practice these functions do lead to collisions with significant probability [25]. This is so because the memory blocks do not change randomly. They change according to a well-defined pattern given by the application program. This increases the probability of collision. Noticing above limitation, secure hash functions were used in [26] to calculate changed blocks. A hash function is called secure if it is computationally very difficult[5] (i.e., infeasible for all practical purposes) to find distinct two blocks B_1 and B_2 such that $H(B_1) = H(B_2)$. Secure hash functions are believed to exist. Hash functions such as MD5,

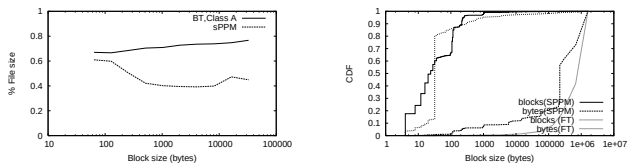
[4]I/O and signal state saving and restoration requires internal book-keeping, including intercepting of several glibc calls using weak aliases.

[5]There are strong theoretical evidences suggesting that there is no algorithm that runs in time O(poly(h)) and finds two such blocks with a probability better than 1/poly(h), where poly(h) is any polynomial in h.



(a) Broad address range (b) Zoom-in

Figure 1: Memory access pattern for sPPM



(a) Effect of block size (b) CDF (unchanged memory)

Figure 2: Deducing optimal block-size

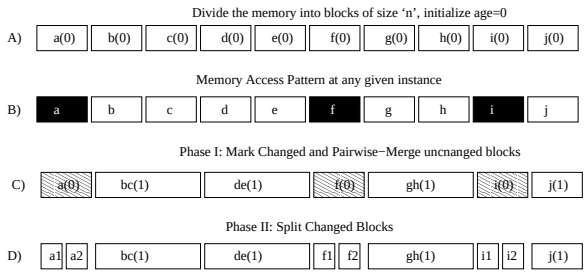


Figure 3: Pictorial view of split-merge algorithm

SHA1 and SHA2 [27] are believed to be secure and are routinely used in cryptography. While deriving strong bounds on the the correctness of this technique (in the context of incremental checkpointing at least) is subject to detailed theoretical and probabilistic analysis, (which is beyond the scope of this paper), sufficient evidence is available in the literature today that confirms that a collision probability of cryptographically secure hash functions like MD5 is as low as order 10^{-19} [27], which is practically negligible. To put this in perspective, probability of a memory bit-flip (due to an alpha particle) is much higher (of order 10^{-7}) [28], which means that an application crash is anyway more likely to occur due to a memory bit flip, than due to a hash collision. Therefore, it is reasonably safe to use secure hash functions for incremental checkpoints.

Choice of suitable block size: There are really two issues to think here: What should be the appropriate block size, and can there be a uniform block size for all applications. Figure 1(a), shows the addresses that are changed (represented by black lines) in the data segment of the sPPM benchmark between two successive checkpoints. It can be seen from the figure that there is a large chunk of unchanged data, from 6.2 MB to 7.8 MB. Further, if we zoom-in on a smaller address range from 2MB - 2.5 MB (as shown in Figure 1(b)), we still see several largely unchanged blocks of data. In practice, the blocks of un-changed data are of different sizes and are placed at arbitrary places.

Intuitively, one would expect that smaller the block size, more is the opportunity to reduce the checkpoint file size, and hence better the performance. However, this may not always be the case as very small blocks may sometimes increase hashing overheads, thereby reducing the overall performance. (See Figure 6(c)). Second one is more tricky. Consider Figure 2(a), which plots the file size of an incremental checkpoint as a function of the block size (in logscale) for two different applications. The file size is shown as the fraction of the basic checkpoint file.[6] The optimal block size for program BT(class A) is 128 bytes while the optimal block size for program sPPM is 4096 bytes ! Similar variations in optimal block sizes have been observed even earlier, and shown in [24].

Clearly, *one-size-fits-all* approach will not work, as different applications can have varying degrees of optimal block sizes. As an example, see Figure 2(b), that shows the cumulative distribution of size of unchanged blocks between two consecutive checkpoints. For the application sPPM, unchanged blocks range from about 10 bytes upto 1.5MB bytes, and for FT, the range is from 100 bytes to over 1MB.

[6]For these plots, we assume a header overhead of 32 bytes per block that is written to the checkpoint file.

More so, within a single application run, changes can be at varying granularities over a period of time, and hence optimal block size can change over time. This serves as a strong motivation to develop a heuristic which could automatically find out the optimal block size, based on memory access patterns of the application(s). The proposed algorithm in the next section exactly outlines one such heuristic.

5. ADAPTIVE BLOCKS BASED INCREMENTAL CHECKPOINT ALGORITHM

We now propose an *adaptive* hash-based incremental checkpoint technique (AIC), which dynamically partitions the application memory into variable sized blocks such that the checkpoint overheads are minimized. The following subsections will discuss the proposed algorithms and various design choices made.

5.1 Checkpointing with variable-sized hash blocks

Allocate a hash table of size n (in unit of entries), for an application using a memory of M bytes. (See subsection 5.3 to understand how to decide n). This allows us to divide entire application memory into n blocks, each of initial block size equal to M/n. With each block, a parameter called *age* is associated, which defines the number of consecutive times a particular block was NOT modified. As shown in the Figure 3 (A), age of each block is initialized to zero. The age tracking mechanism is used to identify blocks which have been un-modified same number of times, and hence they could be merged. The idea of merging them is based on the hope that none of these blocks will change in the near future, or even if they change, they will change together (due to locality of reference principle). The algorithm now works in two phases. **In the first phase**, it computes the hash value of each block of memory and compares it against the value stored in the hash table. If the two values differ, then the corresponding block is marked as dirty (its changed) and

saved into the checkpoint file. Otherwise, if the two values are same, the age of the block is incremented, and all un-changed blocks are scanned to find merge opportunities. A merge can happen for all contiguous un-changed blocks having same 'age'. For instance, in Figure 3, (B) shows changed (black) and un-changed (white) blocks identified in a iteration, then all changed will be marked dirty (grayed, as in Figure 3 (C)) and all un-changed will be merged in pairs of two (again, as in Figure 3 (C)). If no such merge opportunities exist, the algorithm proceeds to next phase. Note that at one instance, no more than 2 contiguous blocks can be merged. This is our 'lazy-merge' optimization, and is explained later in the section 5.3. **In the second phase**, the algorithm sorts the list of changed blocks by size, and starts splitting the largest changed block first, until there is no space left in the hash-table, or the list is empty. For each block that is split, age is reset to 0. Again, see Figure 3 (D), where all changed (grayed) blocks of (C) were split into two. This simple split-merge technique continues at each checkpoint instance, and over a period of time, things stabilize so each changed block is of near-minimum possible size, while each un-changed block is of near-maximum possible size.

5.2 Restarting the application

We have developed a standalone *merge* utility, which would merge all the incremental checkpoint files into one non-incremental checkpoint file. The application can be restarted from this file as if it is started from a regular (non-incremental) checkpoint file. This utility can be used by system administrators to periodically merge various incremental files into a single checkpoint file (offline), thereby reducing on space as well as the time to restart the application. This could either be done manually or by implementing an automatic *garbage collector*. The general algorithm to do so is as follows :-

1. Read the *latest* incremental checkpoint file and write all sections into the final merged file (since *its* all sections are latest).

2. For each subsequent file in reverse order, find address ranges not already written in the final merged file, and copy the corresponding blocks into the final merged file. This ensures only the most recent blocks are written into the final chkpt file.

3. The final merged file thus obtained is the complete non-incremental checkpoint file, ready to be used for restart.

5.3 Key design issues

There are several key design issues and nuances in this approach, that need careful explanation. We discuss each of them as follows :-

What is the optimal hash-table size ? This is probably the most critical factor that decides the possible performance benefits of the proposed AIC approach. Simply put: more the space in hash-table, better is the opportunity for the algorithm to identify exact changed blocks, and vice-versa. The size of this table is usually dependent on how much *extra* memory is available for scratch use in the system, which in-turn depends on what is the application's memory footprint. In our case, in order to do a fair comparison to the fixed-block scheme, we varied the hash-table overhead from 0.1% upto 10% of the total system memory (256MB, for our current system) (See Figures 8, 9, 10 and 11.)

How much to split, atmost ? Blocks are split in order to isolate tightest possible boundaries, but care must be taken not to divide into so small chunks that the header overhead (32 bytes) of the hash-table entry turns out to be more than the actual data. Moreover, one should *intelligently* split, to maximize the potential benefits. If large changed blocks are split, there is potential for greater savings. Therefore, the algorithm splits large changed blocks first, and if space remains, splits the smaller blocks. In any case, we split upto a maximum block size of 32 bytes.

How much to merge at a time ? Merging is tricky. One could be greedy and merge *all* contiguous un-changed blocks at once, hoping to free-up several hash-table entries. But this approach can backfire, if application modifies a large data-structure in alternate iterations. In such a case, at every iteration there is an un-necessary split and merge, and cost is paid in terms of re-hashing time. To see the damage, see Figure 4 (A), showing a few changed (black) and a few un-changed (white) blocks at instance I. Assuming there was no lazy-merge, after going through the first pass, all changed blocks will be split, and all un-changed will be merged, as shown in Figure 4 (B). Now suppose at instance I+1, memory areas **(a,c,f,i)** change, as in Figure 4 (C). All changed blocks (from Figure 4 (B)) will again be split, including the block **bcde**, which was merged in the previous iteration. In next iteration I+2, no area from this chunk was modified again, so it is again merged into **bcde**, as shown in Figure 4 (F). Such a situation easily leads to *thrashing*, as splits and merges happen too fast. We have taken care of this situation by doing a *slow, pairwise* merge, using our *aging* concept (defined earlier). This ensures that even if there is a large number of contiguous unchanged blocks, the algorithm merges them in pairs. For n contiguous unchanged blocks of same age, the algorithm will take $\log(n)$ checkpoints to merge them into a single block.

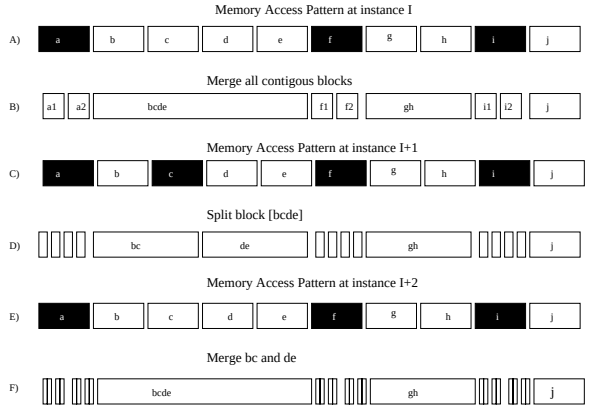


Figure 4: Motivating the need for lazy merge.

How to store changed blocks in checkpoint file? All the *contiguous* changed blocks are written as a single section in the checkpoint file. This optimization eliminates the sec-

tion header overhead for a large number of changed blocks.

Where to store the hash-table itself ? The hash table may either be stored in the memory or written to the checkpoint file. Storing the hash table in memory increases the application memory requirement, while storing the hash table in checkpoint file increases its size and adds to the I/O overhead. If the hash table is stored in the checkpoint file, it needs to be read into the memory at the next checkpoint. This further increases the I/O overhead. Moreover, to avoid adding to the application memory overhead, the hash table needs to be read in small blocks and compared against the memory. This not only increases the complexity of implementation but also degrades I/O performance. We therefore decided to keep the hash table in the memory.

What if this in-memory hash table is lost ? Due to an application crash, it is possible that we may loose the in-memory hash-table. To re-start the application from a previous checkpoint, we do not make use of the hash table, as all the necessary recovery information is coded within the headers of the saved data. Hash table is only used for the checkpointing logic in the AIC/FIC schemes, and it has no role to play at the time of recovery. Hence, even if a crash happens *during* a checkpoint, we will not need the hash-table, as we will simply need to restart from a previous checkpoint. However, in such a restart mechanism, we do loose a bit on performance, as AIC algorithm will have to re-learn the memory access patterns and re-calculate the optimal block sizes, but we believe its worth re-discovering the optimal boundaries (which takes only a few loop iterations) than incurring the overhead of saving the hash table (of order several MB).

6. PERFORMANCE ANALYSIS

This section gives an overview of the experimental setup and a detailed analysis and comparison of different hash-based incremental checkpointing techniques.

6.1 Experimental setup and benchmarks

Our experimental system is a 8-node partition of a BlueGene/L midplane [29, 30]. We have compared the performance of the proposed Adaptive Incremental Checkpoint technique (AIC) with that of the Fixed Incremental Checkpoint technique (FIC) and Non-Incremental Checkpoint technique (NIC) for a variety of benchmarks. At the time of our experimentation, file I/O sub-system of the BlueGene/L machine was still under development and optimization, so we calculated the file write timings by using the *target* bandwidth[7], instead of actual write timings over the NFS to the disk. We chose to report targeted numbers in order to reflect the final performance of the BG/L machine. However, this does not affect the accuracy and correctness of our results, because all the results are scaled-down uniformly.

Our benchmarks come from a variety of sources, including the NAS Parallel Benchmarks (NPB) [31] and the ASCI Purple Benchmarks [32]. We used five out of eight NPB 2.3 benchmarks (**EP, FT, LU, BT, SP**) and two out of seven ASCI Blue Benchmarks (**sPPM, SWEEP3D**). The selection is purely arbitrary, although slightly biased in favor of ease of porting to the BlueGene/L hardware. The code of each benchmark was instrumented to place calls to the checkpoint library. The calls to take a checkpoint were placed in such a way that the checkpoints were taken at regular, uniform intervals during the lifetime of the applications.

[7]Currently planned I/O bandwidth is 40 MB/s per I/O node, resulting in an approximate bandwidth of 333 KBps per compute node, assuming 1 I/O node for 128 compute nodes

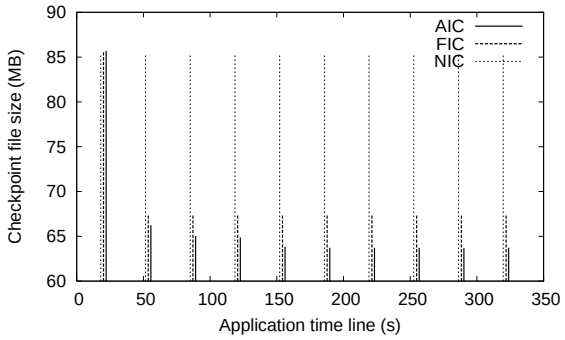


Figure 5: Per-checkpoint reduction (BT, Class A)

APP	No. of ckpts	*Checkpoint file size (MB)*		
		NIC	FIC	AIC
LU (class B)	5	30.2	16.9	14.9
BT (class A)	10	85.3	56.6	55.5
SP (class A)	5	28.5	14.6	13.2
EP (class C)	41	3.0	2.3	1.1
FT (class A)	6	58.4	28.3	26.9
sPPM	7	6.75	4.2	1.8
sweep3D	6	88.8	53.1	52.1

Table 1: Average file size reduction

6.2 Performance metrics and corresponding results

We define four important metrics, for performance evaluation of the proposed AIC scheme against NIC and FIC. While looking at actual results, we discuss the relative importance of these metrics, and quantify the impact of each of the schemes on them.

1. Checkpoint File Sizes : Reducing checkpoint file size is our primary goal, given the anticipated scale of storage requirements on massively parallel systems. For instance, storage needed by an application running on the complete 64K node BlueGene/L machine is of order $16TB$. For NIC the checkpoint file size at a node is essentially dictated by the full memory footprint of the application. For FIC, the file size is governed by (the number of changed blocks) * (block size + size of section header). For AIC, the file size depends mainly on the size of memory present in the changed blocks and the section header overheads for the contiguous changed blocks.

From the Table 1, we can see that both, FIC as well as AIC show a significant reduction in the checkpoint file sizes across all benchmarks. FIC gives a reduction of 22.3-48.7%, while the AIC gives a reduction of 35.0-73.1%. This clearly

APP	App Time(s)	Checkpoint time (s/ckp)		
		NIC	FIC	AIC
LU (class B)	572	90.6	53.1	56.2
BT (class A)	338	256	178.5	195.1
SP (class A)	343	85.5	45.8	49.2
EP (class C)	628	8.7	7.3	3.9
FT (class A)	14	175.3	90.6	100
sPPM	95	20.3	14	6.8
sweep3D	279	264	167	180

Table 2: Average time reduction

shows that AIC is better than FIC, as far as this metric is concerned. We observe a more significant improvement in applications with smaller memory footprints (sPPM, EP), because for the same memory overhead corresponding to a block of 128 bytes, there is more opportunity for them to adapt (smaller application memory size results in lesser hash table entries), as compared to BT, LU and other benchmarks. This suggests that a larger hash-table size will help AIC better. Looking a bit more closer in the Figure 5, for uniformly spaced checkpoints we find that as the application (benchmark BT, class A), progresses, AIC file sizes gets smaller, suggesting that AIC learns the application memory access patterns and optimizes the memory partitioning.

2. Checkpoint Time: From Table 2, we can see that both AIC as well as FIC show significant improvements in terms of overall time reduction, when compared to NIC across all benchmarks. However, AIC takes more time than FIC in many cases, especially for applications with large memory footprints. This can be explained by looking at Table 3, which shows the I/O and hash time overheads for all benchmarks. For both FIC and AIC, the checkpoint time consists of the I/O time and the time to compute the hash of the blocks. For AIC, however, an additional time is required in sorting, scanning and comparing the hash table entries to compute the new partition of the memory. Moreover, for FIC, the hash is computed exactly once for every block, but for AIC, the hash is computed once only for blocks that are not split or merged in the new partition. For every other block of memory, the hash is computed twice. This adds to more hashing overheads, as is clearly seen in Table 3. The I/O times of AIC are generally better than FIC in most cases, which re-iterates the fact that file sizes indeed get reduced. While at present, hash-times might seem like an overkill, we argue that in current systems, I/O is generally a bigger bottleneck than CPU, and hence it is more important to optimize on the I/O performance.

3. Effect of block sizes: First, we analyze the real impact of block-size over the checkpoint performance. For this, we ran several experiments with multiple block sizes, and captured its impact on file size, checkpoint time, I/O time and hash time. Figure 6 show the strong dependence of the performance of the FIC scheme on the block size. Intuitively, smaller block sizes should give better performance for any incremental scheme. But, the overhead associated with block headers and hash time also increases as the block size decreases. This explains the increase in the file size as

APP	I/O time (s)		Hash time (s)	
	FIC	AIC	FIC	AIC
LU (class B)	50.7	44.7	1.4	3.2
BT (class A)	170	170	4.9	7.1
SP (class A)	43.8	39	1.02	2.9
EP (class C)	7.0	3.2	0.1	0.24
FT (class A)	85.7	81.5	2.9	5.6
sPPM	12.6	5.5	0.4	0.6
sweep3D	159	159	4.6	6.8

Table 3: Detailed overheads:I/O and hashing

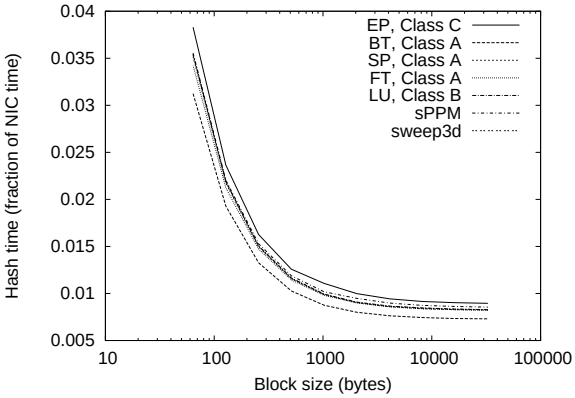
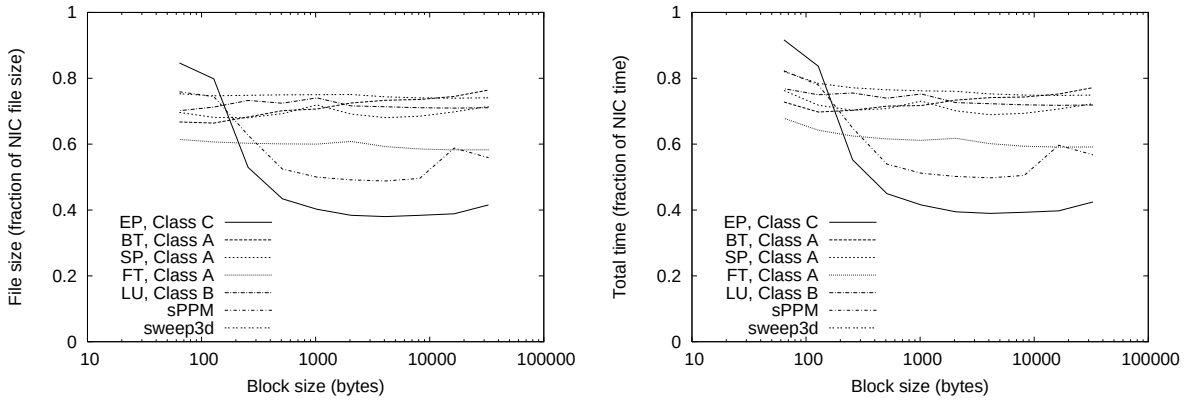


Figure 7: Effect of block-size on hash time

well as checkpoint time for very small sized blocks. The I/O time variation is just proportional to the file size, so we did not plot the graph explicitly. More interestingly, as seen in the Figure 7, as we decrease the block size, the hash time increases monotonically because of the hash overheads associated with each block. This suggests that too much of hashing is deterrent to performance, but if we see absolute numbers, we can observe that the overhead due to hash computation is still less than 4% of the total time for all the benchmarks, so it is not a bottleneck. Overall, the point to bring from the figures is really the fact that each application has a different block-size at which it shows best performance.

With this knowledge, we further experimented to compare the FIC and AIC schemes explicitly. For a fair comparison of the two schemes, we gave them the same amount of *extra* memory (represented as memory overhead in Figures 8, 9, 10 and 11) to be used for hashing.

Figures 8 and 9 show the variation of file size with memory overhead for the FIC and AIC schemes, while Figures 10 and 11 show the variation in total time taken to checkpoint. Overall, we can see that FIC is more sensitive to increase in memory overhead than AIC. This is understandable, as FIC makes no attempt to make intelligent use of extra memory provided. Beyond a point, all increase in the hash table size amounts to unwanted reduction in the *block size* (block size is an inverse function of the memory overhead), which amounts to increased overhead in writing additional data to the disk. On the other hand, in AIC, increase in memory overhead actually reduces the file size, because it provides more opportunity to the split-merge algorithm to



(a) File size (b) Total time

Figure 6: Effect of block-size on all benchmarks for the FIC scheme

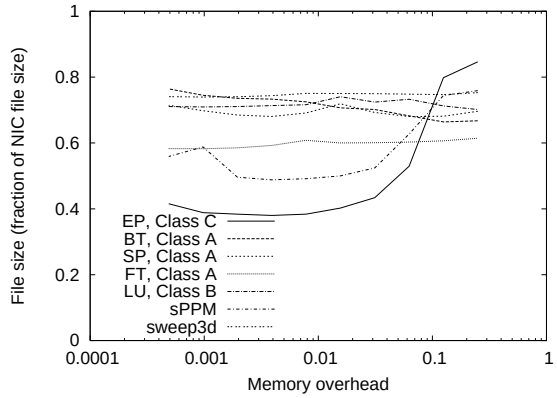


Figure 8: File size variation for FIC

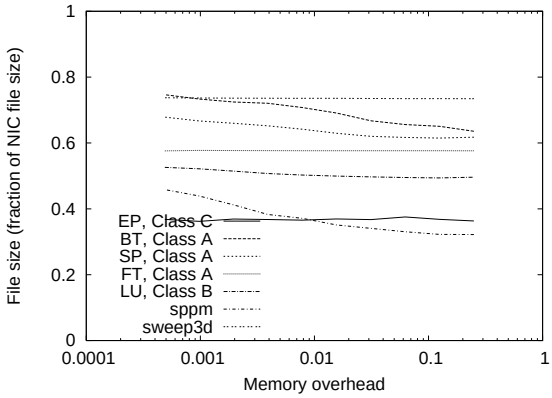


Figure 9: File size variation for AIC

identify finer granularities of changed blocks. In terms of time, however, AIC did not do too good, as the time taken to checkpoint marginally increased in most cases (See Figure 11). As discussed earlier, this is because AIC has an extra overhead of hash computation and optimal memory partitioning, when compared to the FIC scheme.

4. Adaptiveness to memory access pattern: One of the core design principles of our AIC algorithm was to be able to dynamically adapt to the memory access patterns of the applications. We divided the entire memory into initial fixed blocks of 512 bytes, and executed sPPM benchmark with AIC and FIC schemes. Figure 12 shows the cumulative distribution function of blocks of unchanged memory. As shown in the figure, a block size of 512 bytes for FIC does not capture the entire memory pattern, whereas AIC was able to dynamically adapt to the memory access patterns, and came much closer to capturing the exact changed bytes as compared to the FIC scheme.

6.3 Spacing of checkpoints

All the data presented so far was for the programs that do not run for very long. The longest running program was EP, class C which took 628 seconds. As a result, the spacing between two subsequent checkpoints was also very small. In the production environment, the BlueGene/L machine will run jobs that may take several hours, or maybe even days. In such a setting, the checkpoints will be taken at a frequency of several minutes or hours rather than seconds. This raises questions about validity of our results. In particular, if the checkpoints are taken after an hour, it is not clear if the incremental checkpoint technique will still result in similar benefits.

Typically scientific programs carry out large number of iterations in loops. The parts of memory changed by a sequence of iterations of a loop do not change significantly. Therefore, if a call to checkpoint is placed inside a loop, it does not matter if ten iterations of a loop are executed or a thousand iterations of the loop are executed between two successive checkpoints. The amount of memory modified in both the cases, should not to change considerably.

Our preliminary experiments in this direction confirmed our above observations. Figure 13 plots the file size as a function of mean spacing between the checkpoints for the application sPPM. We ran the sPPM program for about 1 hour. For a mean spacing of 16 seconds, 100 checkpoints were taken and the mean file sizes for FIC and AIC were 60% and 48% of NIC file size respectively. Similarly, for a mean checkpoint spacing of 20 minutes, 4 checkpoints were taken and the mean file sizes for FIC and AIC were 61% and 50% respectively. Similar behaviors were observed for other applications including BT, SP and LU. Therefore, it

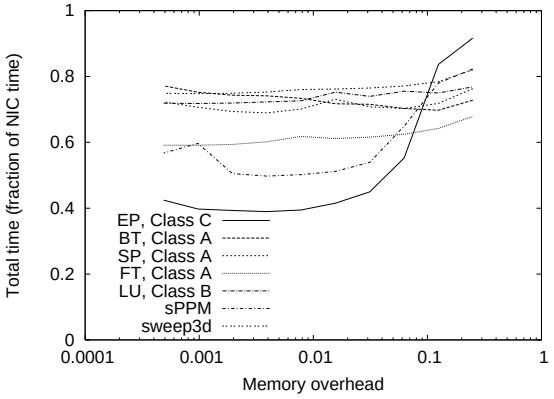


Figure 10: Checkpoint time variation for FIC

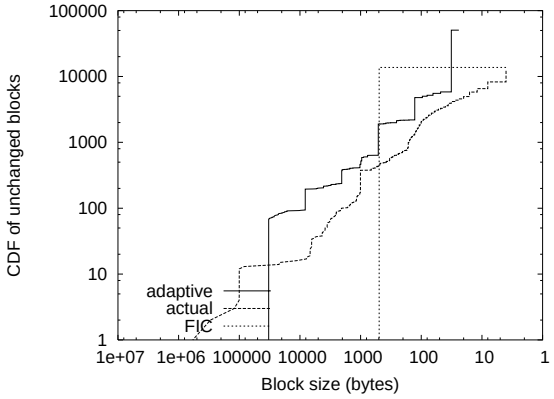


Figure 12: Adaptiveness of AIC

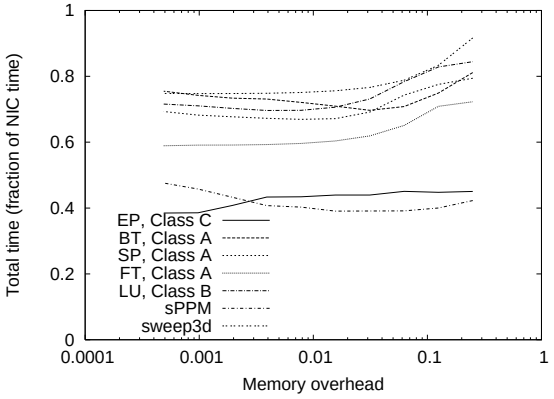


Figure 11: Checkpoint time variation for AIC

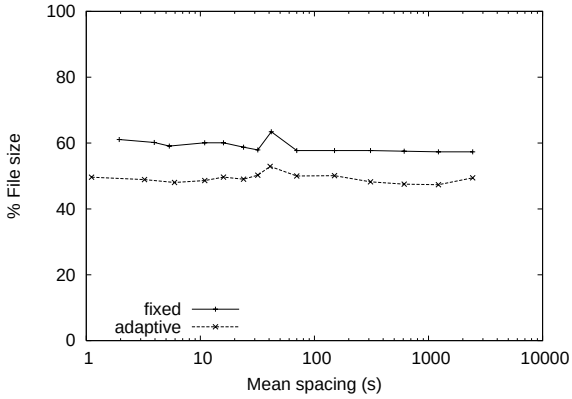


Figure 13: Effect of checkpoint spacing

seems that for most typical scientific applications, spacing between two checkpoints does not have significant impact on the savings obtained by the incremental checkpoint. As a result incremental checkpoint techniques can significantly reduce the checkpoint overheads on the BlueGene/L machine.

7. CONCLUSIONS

In this paper, we have shown that within the scope of the state-of-the-art technology, a page-based approach for incremental checkpointing may not be capable of addressing the unique scalability and self-optimization challenges, typical on very large systems like BlueGene/L. We did a study of memory access patterns of various parallel applications, and based on our observations we have proposed a new, adaptive incremental checkpoint technique which is capable of dynamically adjusting the block sizes to capture *near-exact* changed parts of the memory. Core to this design is a novel *split-and-merge* algorithm that keeps track of application's memory access history, in order to determine which blocks to split and which ones to merge.

Our experimental results indicated upto 70% improvements with respect to the file size reduction and upto 50% reduction in the time taken for a checkpoint, when compared to non-incremental checkpoint technique. We also observed upto 25% improvements with respect to file size reduction when compared to the fixed incremental checkpoint technique, but we experienced a 5%-10% loss with respect to time taken to checkpoint.

8. FUTURE DIRECTIONS

While we believe that this is the first attempt on designing an adaptive and self optimizing scheme for incremental checkpointing, we admit that there is still significant room for further optimizations to obtain better performance. There are several directions to look at, in future. For instance, tests on a bigger system (of order of several hundreds of nodes) are required to further validate the approach. Real measurements on an actual file system are still required to validate our results. Moreover, a larger file system bandwidth might magnify the hashing overheads. A more detailed evaluation would be necessary in that scenario to see the benefits of the AIC scheme over the FIC scheme. While the focus of our current work was to come up with a novel checkpointing technique for our experimental platform: BG/L, we would like to perform a comparative analysis of AIC vs traditional page-based incremental checkpointing approach on a Linux cluster, and evaluate the feasibility of our technique on such platforms. Finally, the checkpoint library needs to be system-initiated rather than application-initiated, in order to be truly adaptive.

9. REFERENCES

[1] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *IEEE Transactions on Computers*, vol. 46, Aug. 1997.

[2] J. S. Plank and W. R. Elwasif, "Experimental assesment of workstation failures and their impact on checkpointing systems," in *28th International Symposium on Fault-Tolerant Computing*, June 1998.

[3] J. S. Plank, J. Xu, and R. H. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," Tech. Rep. CS-95-302, University of Tennessee at Knoxville, Aug. 1995.

[4] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, Aug. 1994.

[5] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[6] Princeton University Scalable I/O Research, "A checkpointing library for Intel Paragon." http://www.cs.princeton.edu/sio/CLIP/.

[7] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, pp. 213–223, Jan. 1995.

[8] M. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message passing systems," Tech. Rep. CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 1996.

[9] Y. Chen, J. S. Plank, and K. Li, "CLIP: A checkpointing tool for message-passing parallel programs," in *SC97: In Proceedings of the Supercomputing*, Nov. 1997.

[10] A. Beguelin, E. Seligman, and P. Stephan., "Application level fault tolerance in heterogeneus networks of workstations.," *Journal of Parallel and Distributed Computing*, vol. 43, pp. 147–155, May 1997.

[11] D. Pei, D. Wang, and Y. Zhang, "Quasi-asynchronous migration: A novel migration protocol for PVM tasks," *ACM Operating Systems Review*, vol. 33, Apr. 1999.

[12] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of mpi programs.," in *In Principles and Practice of Parallel Programming*, June 2003.

[13] C. J. Li and W. K. Fuch, "CATCH - Compiler assisted techniques for checkpointing.," in *In Proceedings of the Internationnal Symposium on Fault Tolerant Computing*, June 1990.

[14] A. Agbaria and J. S. Plank, "Design, implementation, and performance of checkpointing in netsolve.," in *In Proceedings of the International Conference on Dependable Systems and Networks.*, June 2000.

[15] J. S. Plank, Y. Kim, and J. J. Dongarra, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 972–986, Oct. 1998.

[16] J. S. Plank and K. Li., "ickp: A consistent checkpointer for multicomputers.," *IEEE Parallel and Distributed Technologies*, vol. 2, pp. 62–67, June 1994.

[17] J. Pruyne and M. Livny, "Managing checkpoints for parallel programs.," in *In Proceedings of the IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 1996.

[18] E. Pinheiro, "Truly-transparent checkpointing of parallel applications." http://www.cs.rutgers.edu/ edpin/epckpt/paper_html/.

[19] S. Sankaran, J. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The design and implementation of Berkeley lab's Linux checkpoint/restart," in *Los Alamos Computer Science Institute (LACSI) Symposium*, Oct. 2003.

[20] H. Zhong and J. Nieh, "CRAK: Linux checkpoint/restart as a kernel module," Tech. Rep. CUCS-014-01, Department of Computer Science, Columbia University, Nov. 2001.

[21] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors.," in *In Proceedings of the International Symposium on Computer Architecture*, May 2002.

[22] D. J. Sorin, M. K. Martin, M. D. Hill, and D. A.Wood, "Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *In Proceedings of the International Symposium on Computer Architecture*, May 2002.

[23] J. C. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg, "On the feasibility of incremental checkpointing for scientific computing," in *International Parallel and Distributed Processing Symposium*, (Santa Fe, NM, USA), April 2004.

[24] H. Nam, J. Kim, S. J. Hong, and S. Lee, "Probabilistic checkpointing," *IEICE Transactions, Information and Systems*, vol. E85-D, July 2002.

[25] E. Elnozahy, "How safe is probabilistic checkpointing," in *Twenty Eight Annual International Symposium on Fault-Tolerant Computing*, pp. 358–363, 1998.

[26] H. Nam, J. Kim, S. J. Hong, and S. Lee, "A secure checkpointing system," *Journal of Systems Architecture*, vol. 48, pp. 237–254, 2003.

[27] A. J. Menezes, P. C. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. 1997.

[28] SUN Microsystems Inc.,, "Soft memory errors and their effect on sun fire system." http://www.sun.com/products-n-solutions/hardware/docs/pdf/816-5053-10.pdf.

[29] N. Adiga and et. al., "An overview of the BlueGene/L Supercomputer," in *In Proceedings of the Supercomputing*, Nov. 2002.

[30] G. Almasi, R. Bellofatto, J. Brunheroto, C. Ca—scaval, J. G. Castaqos, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, , and K. Strauss, "An Overview of the BlueGene/L System Software Organization," in *Euro-Par: 9th International European Conference on Parallel Processing*, Aug. 2003.

[31] D. Bailey, T. Harris, W. Saphir, R. vander Wijngaart, A. Woo, and M. Yarros, "The NAS parallel benchmarks 2.0," Tech. Rep. NAS-95-020, NAS Systems Division, Dec. 1995.

[32] "ASCI blue benchmarks." http://www.llnl.gov/asci_benchmarks/asci/asci_code_list.html.