On the Feasibility of Incremental Checkpointing for Scientific Computing

José Carlos Sancho Fabrizio Petrini Greg Johnson Juan Fernández
Eitan Frachtenberg

Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory, NM 87545, USA
{jcsancho,fabrizio,gjohnson,juanf,eitanf}@lanl.gov

Abstract

In the near future large-scale parallel computers will feature hundreds of thousands of processing nodes. In such systems, fault tolerance is critical as failures will occur very often. Checkpointing and rollback recovery has been extensively studied as an attempt to provide fault tolerance. However, current implementations do not provide the total transparency and full flexibility that are necessary to support the new paradigm of autonomic computing – systems able to self-heal and self-repair. In this paper we provide an in-depth evaluation of incremental checkpointing for scientific computing. The experimental results, obtained on a state-of-the art cluster running several scientific applications, show that efficient, scalable, automatic and user-transparent incremental checkpointing is within reach with current technology.

Keywords: *Fault-tolerance, Checkpointing, Autonomic Computing, Rollback Recovery, Large-scale Parallel Computers, Performance Evaluation.*

1 Introduction

Thanks to the easy availability of powerful commodity processors and networks it is now possible to construct very large computing clusters.[1] Unfortunately the large number of components in such clusters results in a extremely high combined failure rate.

For example, the proposed BlueGene/L [2] system with 65,536 processors is expected to experience failures every few hours. Because these computers are intended to be capability engines executing demanding scientific applications, such as the ASCI stockpile certification programs, there is a high probability of hardware or software failure during program execution. Moreover, the increasing complexity of these clusters means that the system may still fail despite the increase in hardware reliability. Thus, mechanisms to tolerate faults have become a critical design point in such systems.

Employing redundancy of the components to tolerate failures, such as using replication of processors, memories, and interconnects is not a cost-effective solution for commodity clusters because of the highly competitive commercial market that seeks lower cost solutions. However, checkpointing and rollback recovery is a promising approach to tolerate failures due to its simplicity and lower cost. Checkpointing and rollback recovery is based on periodically saving the process state to stable storage. Then, in the event of a failure, the application can be rolled-back from the most recent checkpoint to restart the execution as if the fault had never occurred.

Since commodity clusters are being used as capability platforms, system availability and efficiency are increasingly important requirements. The high failure rate of these systems puts more pressure on checkpoint mechanisms. In order to meet these requirements, checkpoints should be taken frequently relative to the failure rate of the system, for example every few minutes.

Moreover, the increasing complexity of managing and maintaining clusters brings a new demand for system dependability. There is an inevitable need for autonomic computing systems which are able to self-heal and self-repair [16]. To achieve this vision, checkpoint and rollback recovery mechanisms must be automatic– they should work without the intervention of programmers, users, or system administrators.

In this paper we analyze several parallel scientific applications written in Fortran using the MPI communication library targeted to a cluster architecture. Our primary objective is to answer the following question. Is it possible to implement an incremental checkpointing system that is completely automatic and user-

[1] See www.top500.org.

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

transparent, minimally intrusive, scalable and feasible with current and foreseeable I/O technology?

Our major contribution is to demonstrate that *frequent, user-transparent, automatic incremental checkpointing is a viable technique*. We also prove that this can be achieved without using specialized hardware and within the limitations imposed by current technology. We anticipate that technological advances in networking and secondary storage will further enable these methods.

The specific contributions of this paper are the following.

Limited Bandwidth Requirements For the applications we analyzed, the bandwidth required for incremental checkpointing is of the same order as today's high performance interconnection networks and secondary storage arrays provide. Checkpointing intervals of a few seconds are possible with current technology, and we argue that shorter intervals will be possible in the future.

Periodic Application Behavior We have identified bulk-synchronous patterns of behavior in these applications that can be exploited to implement efficient checkpointing algorithms. These codes typically alternate between processing and communication bursts that can automatically be identified at run time, for example using global operators such as the STORM mechanisms [11]. This behavior can be exploited to implement efficient coordinated checkpoints.

Scalability of Results We show that the bandwidth required per process decreases as cluster size increases (assuming weak scaling of the application). Furthermore, the bandwidth requirements grow sublinearly with the size of the memory footprint.

The rest of this paper is organized as follows. In Section 2 we describe an important dimension of the design space of checkpoint and rollback recovery systems. Section 3 specifies the goals we wish to achieve in this paper. Section 4 describes our experimental methodology used to analyze scientific applications. We describe the experimental environment in Section 5. Section 6 describes the experimental results. Section 7 summarizes related work on checkpointing. And finally, Section 8 provides some concluding remarks.

2 Design Space of Checkpoint and Rollback-Recovery Systems

Many solutions have been developed in the past decade to provide fault tolerance based on checkpoint and rollback-recovery. An excellent survey of these techniques can be found in [10].

One important dimension of the design space is the abstraction level at which the state of the process is saved. This dimension will be analyzed in some detail in the following section.

2.1 Abstraction Level

For the sake of our discussion, we consider four abstraction levels at which incremental checkpointing can be implemented. The highest level of abstraction correspond to the application level, in which the checkpointing is implemented inside the application. Lower levels of abstraction corresponds to implementations in a run-time library, in the operating system, or through the support of special hardware.

Application Level This approach relies on the application to provide its own fault tolerant capabilities. There are two main approaches described in the literature. One is based on mathematical properties of some scientific applications which converge to the correct result even in the presence of system failures [12]. The applicability of this method is obviously limited, since only some applications meet its requirements. The other approach is based on modifying the application's source code to perform checkpoint/recovery [6]. The programmer directly inserts checkpoint/recovery points into the program's source code with the support of a library that implements checkpointing primitives. A refinement of this approach is to use the compiler to automatically insert the checkpoint code in a way that is nearly transparent to the programmer [15].

Run-time Library Level A run-time library implements the checkpoint and recovery mechanisms without substantial modification of the application's code [17]. The checkpoint library sets an alarm to periodically interrupt the application and records the process state through the use of system calls. In many cases the run-time library's use of signal handlers interferes with application or the resource manager. Also, it may not be possible to write-protect some segments of the address space, such as the stack, or the network receive buffers in direct-access high-performance networks.

Operating System Level This checkpoint mechanism is implemented inside the operating system kernel [7]. There are many details of a process's state which are only known to the kernel or are otherwise difficult to re-create outside the kernel, such as the status of open files and signal handling. The main advantage of this approach is that it is totally user-transparent and requires no changes to any application code.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Table 1. Comparison of the Checkpointing Abstraction Levels

Level	Transparency	Portability	Checkpoint size	Flexibility of Checkpointing Interval	Granularity
Application with library support	Low	High	Low	Low	Data Structure
Application with compiler support	Medium	High	Medium	Low	Data Structure
Run-time library	Medium	Medium	High	High	Memory Segment
Operating system	High	Low	High	High	Memory Page
Hardware	High	Very Low	High	High	Cache line

Hardware Level This checkpoint scheme is implemented with the support of special hardware [22, 25]. As with operating system level implementations, this method is totally transparent to the users. But hardware-level checkpoints have limited applicability, because they rely on special hardware that is unavailable on commodity clusters composed of off-the-shelf components.

Table 1 provides a comparison between the various checkpointing abstraction levels. The comparison takes into account several metrics including transparency to the user, portability to other environments, the size of the checkpoint data, the flexibility to adjust the checkpoint intervals, and the granularity of the checkpoint.

The major advantages of the application level checkpoints are high reduction in the size of the checkpoint data and the high portability of the checkpoint files that can migrated across heterogeneous machines. Application level checkpoints can exploit the knowledge of the programmer and compiler to insert checkpoints at the best places and to exclude some irrelevant data in order to reduce the size of the checkpoint. This is inconvenient with lower abstraction levels since the application's semantics cannot be used.

The major disadvantages of the application level checkpoints are the lack of transparency and flexibility. Instead of providing automatic mechanisms to manage fault tolerance, users or programmers are needed in order to recompile or relink the application's source code. This approach forces the task of recovering from a failure onto the programmer. Since scientific applications are typically many thousands of lines of code, it is time consuming and error-prone to retrofit fault-tolerance onto an existing code. In addition, in many cases the application source code may not be accessible to the programmers.

Moreover, the checkpoint interval is determined by the application execution, since the checkpoints are only inserted at specified points in the program. This poor flexibility can cause a low efficiency of the cluster under failures.

Providing fault tolerance at the operating system level or the hardware level is a convenient approach, since it provides a higher level of transparency and flexibility allowing automatic checkpoint and recovery without user intervention. Unfortunately, these approaches lack the semantic information available at the application level, so the size of the checkpoint is consequently larger.

In this paper we show that even without this high-level information, checkpointing at the operating system level can discover important properties of the applications at run time and can provide an attractive compromise between productivity and performance.

3 Aims and Limitations

Checkpoint and rollback recovery is a promising approach to support the new computing paradigm of autonomic computing. However, the main challenge of this approach is to efficiently save the potentially large checkpoint data.

There are two potential bottlenecks to saving the checkpoint data: (1) the path over which the data must travel (system bus and the interconnection network), and (2) the storage device on which the data will be stored (memory and/or hard disks). Actually, the network and the hard disks have been considered the critical components due to the lowest available bandwidth [19]. Currently, the new Elan4 QsNet II [1] network and the SCSI hard disks [24] provide a peak bandwidth of 900MB/s and 320MB/s, respectively.

The goal of this paper is to prove the feasibility of checkpoint and rollback recovery with total transparency and full flexibility and without requiring changes to the application code or the support of special hardware. To demonstrate the feasibility of checkpoint and rollback recovery, we are not testing a full checkpoint implementation, but we have constructed a run-time library that is able to quantify the bandwidth requirements needed to save the checkpoint data. By comparing the required bandwidth with the bandwidth available, we will determine the feasibility of implementing a checkpoint mechanism.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

In particular, the following goals are addressed in this paper to show the feasibility of checkpointing:

- **Characterization of Scientific Applications**. We analyze the memory footprint of parallel scientific applications and how the state of that memory changes over time.
- **Bandwidth Quantification**. Measure the bandwidth requirements to save the memory footprint every checkpoint interval.
- **Sensitivity Analysis**. Analyze the influence on the bandwidth requirements of the memory footprint size of the applications, the checkpoint interval, and the number of processors in the cluster.

4 Experimental Methodology

In order to analyze the memory footprint of scientific applications, we have implemented a user-level instrumentation library. This library is preloaded by the dynamic linker (via the LD_PRELOAD environment variable), so that modification or recompilation of the application is unnecessary.

4.1 Memory Footprint of UNIX Processes

To checkpoint a process, it is necessary to save the process's state. The state of a UNIX process generally consists of its address space which is divided into text, data, and stack, and also the information related to the process maintained by the kernel (such as status of open files, signals, and registers). For the sake of simplicity, we consider only the data region of the process, because it represents the largest part of the process's state.

The data memory is composed of four areas – initialized data, uninitialized data, the heap, and mmap'ed memory. The initialized data is data whose value is set at compile time. Uninitialized data is memory allocated at compile time which will be zero-filled by the kernel at load time. The heap is a dynamic area of memory allocated at run time by the *brk* and *sbrk* system calls. Mmap'ed memory is also dynamically allocated and deallocated at run time by the system calls *mmap* and *munmap*, respectively. Generally, the dynamic memory varies in size during the execution of the application due to the allocation and deallocation of memory blocks. The utilization of dynamic memory depends on the compiler. The Intel Fortran77 compiler allocates dynamic memory to the heap, while the Intel Fortran90 compiler uses both the heap and the mmap memory areas.

The layout of the address space in memory depends on the processor and operating system. As a example, for the Itanium II processor, the initialized and uninitialized data follow the text then the heap, which grows toward higher addresses. The address of the top of the heap can be determined by the *sbrk* system call. The stack starts at a fixed address and grows down toward lower positions. Also, for mmap'ed memory, the instrumentation library intercepts the *mmap* and *munmap* system calls that are called by the application to keep track of their boundaries and size.

4.2 Monitoring the Memory Activity

The library uses the memory protection mechanism of the virtual memory system to keep track the pages written to by the process during a certain interval of time called *timeslice*. The protection of each page of memory is set to *read-only*. When the processor attempts to write to a protected page, the operating system sends the process a *SEGV* signal. The library installs a signal handler for the *SEGV* signal which keeps track of pages in which the write accesses occur, called *dirty pages*. The page is then unprotected so that future writes to it in that timeslice do not cause segmentation faults.

The instrumentation library also sets an alarm in order to periodically record the memory footprint and the number of dirty pages. The alarm handler also resets the count of dirty pages and re-protects all the pages belonging to the data memory region space.

The instrumentation library is not able to write-protect the stack, because the stack must be modifiable when executing the signal handler. However, this limitation has an insignificant influence on the results, since as stated above, the data memory accounts for the largest share of the process's state. The maximum stack size measured in our experiments is less than 42 KB.

The *Quadrics QsNet* network interface generates some problems in our instrumentation library, because of its ability to directly access user-space memory. The problems arise when the NIC attempts to write into memory that has been write-protected by the *mprotect* system call. To work around this issue, the instrumentation library intercepts the process's network receive calls so that messages from the network can be received into a buffer that has not been write-protected. The library then copies the message to the proper location in the process, causing segmentation faults for pages that have not yet been written, introducing an unavoidable overhead.

The library also intercepts the call to *MPI_Init()* in order to initialize all the data structures of the library, set up the signal handlers for the timer and the *SEGV* signal, and to write-protect all the process's data memory space.

Because the size of the dynamic memory space changes during the program's execution, the library only reports the dirty pages belonging to the current

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

memory size at the time of the alarm interrupt. The pages belonging to unmapped areas are not taken into account because they will not be used by the application in the future, so there is no need to checkpoint these pages. This allows the use of memory exclusion optimizations [18] which can substantially reduce the size of the checkpoint for some applications.

5 Experimental Environment

Our evaluation was performed on a Linux cluster of 32 HP Server rx2600 computers connected by the *Quadrics QsNet* network. Each HP Server rx2600 contains two Itanium II processors with two PCI-X I/O busses. The Itanium II has a higher memory bandwidth than other modern processors such as the Alpha Ev67 and the Pentium Xeon [26]. The processor with the largest memory bandwidth represents the worst case for incremental checkpointing, because it is able to write a larger amount of memory per unit of time. Therefore the results obtained on the Itanium II can be easily generalized to slower processors.

The scientific applications used in the evaluation are parallel programs written in Fortran and use MPI for inter-processor communications. We have used Sage, Sweep3D, and the NAS Parallel Benchmarks suite [5]. Sage [14] is a large-scale parallel code written in Fortran90 and is representative of the ASCI workload. Sweep3D [27] is a benchmark code written in Fortran77 that represents the heart of a real scientific application. The NAS Parallel Benchmark suite [5] is a set of Fortran77 programs extensively used to evaluate the performance of parallel supercomputers. In particular, we have used the FT, LU, BT, and SP benchmarks.

These applications exhibit different memory allocation patterns. Sage dynamically allocates and deallocates a large part of its data structures, while both the NAS parallel benchmarks and Sweep3D statically allocate their data.

The memory footprint size for these applications is summarized in Table 2. In the NAS parallel benchmark suite we use the problem sizes corresponding to NAS *class C*, for Sweep3D we use a problem size of 1000×1000×50 grid points, and for Sage we vary the problem size in order to evaluate different memory footprints. The memory footprint size is controlled by specifying in the input deck the number of cells per processor. We consider footprint sizes of approximately 50MB, 100MB, 500MB, and 1000MB. For all the applications except Sage, the memory size remains constant during the execution of the application.

We have run each experiment several times in order to minimize fluctuations in our performance measurements. The results presented are the mean values across all executions omitting the first one, because the first experiment takes considerably longer time due to the disk cache misses.

Table 2. Memory Footprint Size (MB)

Application	Maximum	Average
Sage-1000MB	954.6	779.5
Sage-500MB	497.3	407.3
Sage-100MB	103.7	86.9
Sage-50MB	55	45.2
Sweep3D	105.5	105.5
SP	40.1	40.1
LU	16.6	16.6
BT	76.5	76.5
FT	118	118

6 Experimental Results

In this section, we define our performance metrics and identify some important properties of the applications described in Section 5 in order to show the feasibility of incremental checkpointing.

6.1 Performance Metrics

Some of the scientific applications described in Section 5 have been the subject of in-depth performance evaluation and modeling studies [14, 27]. These studies provide insight into the scaling properties, the communication patterns and the expected time to solution on a wide variety of parallel architectures.

In this section we look at these applications from a different perspective: we focus our attention on the performance aspects that are essential to perform incremental checkpointing. In order to do that, we first define an input parameter, the checkpoint timeslice, and two performance metrics that will help us in the performance evaluation.

Checkpoint Timeslice We assume that a global checkpoint is taken across all processes belonging to a parallel program at regular intervals, which we call *checkpoint timeslice*. In our analysis we will assume that the checkpoint timeslice is fixed during the execution of a program.

Incremental Working Set Incremental checkpointing [17] is a cost-effective strategy when the size of the *delta*, the subset of the address space that needs to be saved at the end of a checkpoint timeslice, is relatively small when compared to the whole address space. An important metric to determine the feasibility of the incremental checkpoint is the *Incremental Working Set* (IWS), the set of pages that are written in a timeslice.

Incremental Bandwidth Another metric, derived directly from the IWS, is the *Incremental Bandwidth* (IB), and is computed as the ratio between the

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

size of the IWS and the timeslice length. The IB describes the basic bandwidth requirements that incremental checkpointing algorithms must face.

In our analysis, we are going to present some quantitative and qualitative properties of the IWS and IB, for the applications listed in Section 5. Given that all these applications display a bulk-synchronous behavior with similar performance characteristics on each process, in most cases the behavior of a single process (and, consequently a single graph) is able to capture the behavior of the entire parallel program. All the experimental results in this section use the largest configuration of our cluster, 64 processors, unless otherwise specified.

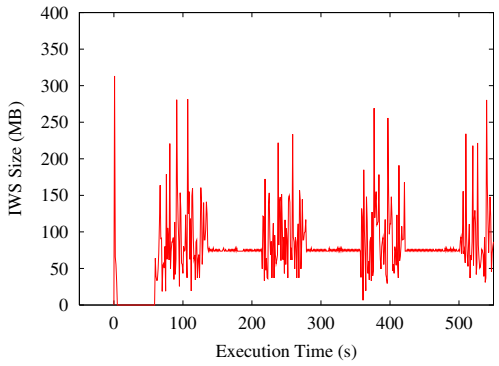
6.2 Relevant Properties of Scientific Applications

Many scientific applications display regular computational and communication patterns. For example Figure 1(a) shows how the IWS size varies when running Sage-1000MB. The initial peak at the very beginning of the execution, is caused by data initialization. After that, we can easily identify a regular pattern, with write bursts every 145s. We will refer to the write bursts as *processing bursts*. Inside the processing bursts, the pages of the IWS are accessed multiple times. Also, the communication takes place in bursts, which we define as *communication bursts*. To illustrate these communication bursts, Figure 1(b) shows the size of the data received in each timeslice by Sage-1000MB. Usually, the communication bursts are placed between the processing bursts. A similar behavior can also be observed in Sweep3D, FT, LU, SP, and BT, but for the sake of brevity the graphs are not plotted.

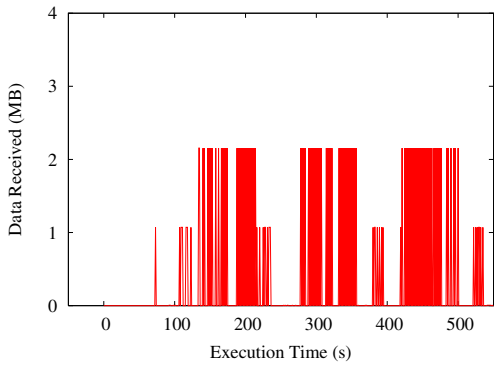
The reason for this regular behavior is that scientific codes perform a sequence of similar iterations, and in each iteration we can identify regular computation and communication bursts. The gap between processing bursts usually identifies the duration of the main iteration of these codes.

Table 3 describes the duration of the main iteration for each application. Among all the applications, Sage executes the longest iterations, while the NAS benchmarks have the shortest ones. In the same table, we can observe that the iterations increase their run time with the data set size. In particular, for Sage the iteration ranges from 20s, with a memory footprint of 50MB, up to 145s for 1000MB. Larger memory footprints increase the duration of both processing and communications bursts due to the increased amount of interprocess communication.

Given this two-phase nature of the scientific codes, there are moments where it is more convenient to take a checkpoint, for example at the beginning or at the end of an iteration. On the other hand, it may not be convenient to checkpoint during a processing burst, be-



(a) Sage-1000MB, IWS Size



(b) Sage-1000MB, Data Received

Figure 1. IWS size and data received per timeslice during the execution of Sage-1000MB, timeslice 1s.

Table 3. Characteristics of the Main Iteration

Application	Average Period (s)	Percent of Memory Overwritten
Sage-1000MB	145	53%
Sage-500MB	80	54%
Sage-100MB	38	56%
Sage-50MB	20	57%
Sweep3D	7	52%
SP	0.16	72%
LU	0.7	72%
BT	0.4	92%
FT	1.2	57%

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

cause pages are likely to be re-used in a short amount of time.

Moreover, the memory written during an iteration represents a substantial subset of the total data set. Table 3 lists the fraction of memory footprint written during a processing burst. Some applications, like BT, practically update the whole memory image (92%), while Sage overwrites only about 55%.

6.3 Bandwidth Requirements

In this section, we quantify the bandwidth requirements for checkpointing the applications described in Section 5. We report the maximum and the average IB collected at run time. In the experiments we study the impact of the checkpoint timeslice on the IB by using timeslices in the range from 1s to 20s. In the analysis we will not consider the write burst at the very beginning of the execution related to the data initialization.

Figure 2(a) shows the bandwidth required to support incremental checkpointing with various timeslices for Sage-1000MB. As the length of the timeslice increases (or, equivalently, the frequency of checkpointing decreases), the checkpointing bandwidth decreases. In particular, the average bandwidth requirements ranges from 78.8MB/s with a timeslice of 1s down to 12.1MB/s with 20s. With a longer timeslice, the pages of the IWS tend to be reused, leading to a reduction in bandwidth.

Similar results are also obtained for Sweep3D, the NAS benchmarks, BT, SP, FT, and LU which are shown in Figures 2(b), 2(c), 2(d), 2(e), and 2(f), respectively. The average and the maximum IB obtained for these applications are practically equivalent because the timeslices used are longer than the duration of the main processing bursts (see Table 3).

Short timeslices corresponds to the worse case to checkpoint the applications because they demand more bandwidth. Table 4 shows the maximum and average bandwidth requirements for all the applications with a timeslice of 1s. It is interesting to note that, even with 1s, the maximum IB is lower than the bandwidth provided by state of the art high performance networks (900MB/s, see section 3), and secondary storage (320 MB/s, see section 3). In particular, Sage-1000MB, the most demanding application for our analysis, requires on average only 78.8 MB/s, 9% of the available peak network and 25% of the peak disk bandwidth.

6.4 Scalability

In this section, we analyze the impact of two important parameters, the number of processors and the memory size of the application. The goal is to provide an expectation of performance on larger machines equipped with more memory, in order to generalize our results to more powerful, future parallel computers.

Table 4. Bandwidth Requirements (MB/s)

Application	Maximum	Average
Sage-1000MB	274.9	78.8
Sage-500MB	186.9	49.9
Sage-100MB	42.6	15
Sage-50MB	24.9	9.6
Sweep3D	79.1	49.5
SP	32.6	32.6
LU	12.5	12.5
BT	72.7	68.6
FT	101	92.1

6.4.1 Increasing the Memory Footprint Size

Figure 3 shows the average IB for Sage with different memory footprint sizes. The IB increases with the memory size due to the larger data structures that are managed in larger working sets. However, the increment of the IB is sublinear. The average IB for Sage-500MB is roughly 50MB/s with a timeslice of 1s, while for Sage-1000MB is 80MB/s instead of 100MB/s. This is because the ratio between the IWS and the whole memory image decreases as we increase the memory footprint, as can be seen in Figure 4.

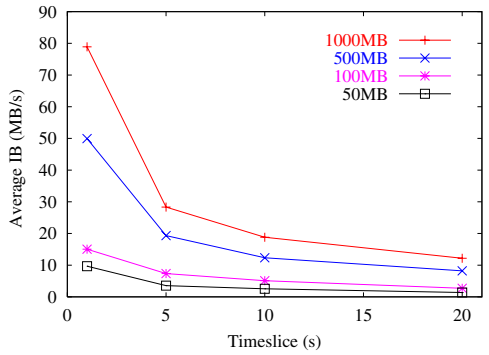


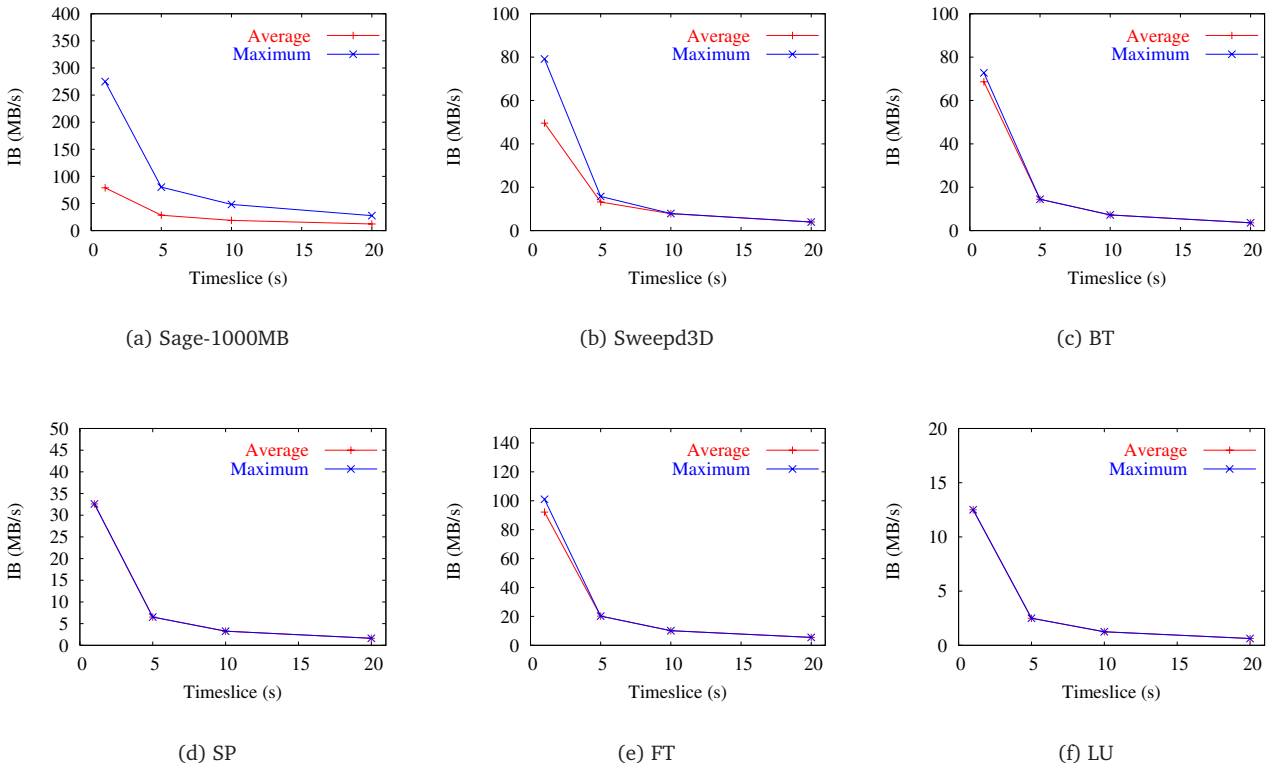
Figure 3. Average IB for Sage-50MB, Sage-100MB, Sage-500MB and Sage-1000MB.

6.4.2 Increasing the Number of Processors

In order to assess the impact of the number of processors we use weak scaling —the problem size grows proportionally with the number of processors— with each processing element doing approximately the same amount of work.

Figure 5 shows the IB for Sage-1000MB with 8, 16, 32, and 64 processors. As can be seen, the number of

0-7695-2132-0/04/$17.00 (C) 2004 IEEE



(a) Sage-1000MB (b) Sweepd3D (c) BT

(d) SP (e) FT (f) LU

Figure 2. Maximum and minimum IB required for checkpointing the applications.

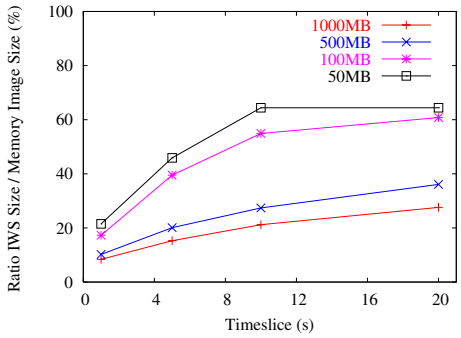


Figure 4. Ratio of IWS size to memory image size per timeslice for Sage-50MB, Sage-100MB, Sage-500MB, and Sage-1000MB.

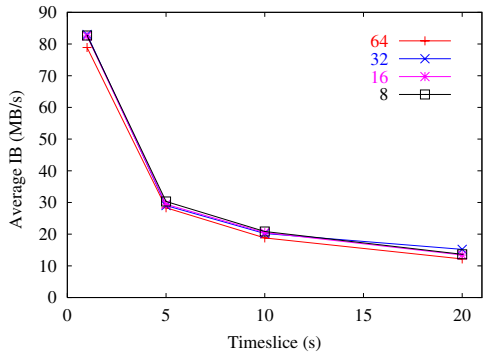


Figure 5. Average IB for 8, 16, 32, and 64 processors for Sage-1000MB.

processors doesn't have a significant influence on the IB. Actually, when we increase the number of processors, the per-processor IB is slightly lower. We argue that this is an important contribution: the results presented here can be generalized to larger computers.

6.5 Intrusiveness

The instrumentation that we have used in the experiments inevitably causes a slowdown in the application run time. We have assessed the overall impact of our measurements for Sage-1000MB reporting a slowdown lower than 10% for a timeslice of 1s. Most of the overhead is caused by the page fault handler that keeps track of the write accesses to memory. Moreover, when we increase the timeslice the impact of the page fault handler is mitigated by the data reuse decreasing the intrusiveness introduced in our measurements.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

6.6 Technological Trends

The performance of the parallel scientific applications is sensitive to the main memory performance. Technological trends have produced a large and growing gap between processor and main memory speeds. Specifically, the performance of the processors is growing a 60% per year, while the main memory only shows a 7% improvement per year [13]. In all, the performance of scientific applications doubles approximately every two to three years. On the other hand, both networking and storage technologies continue to improve at a faster pace. For example, 10GB/sec Infiniband network interfaces will be available by 2005.[2] So we expect that in the near future the increasing bandwidth will make incremental checkpointing even more effective.

7 Related Work

There has been much theoretical work in the field of distributed fault-tolerance using coordinated checkpointing and rollback recovery. However, only a few systems have actually been implemented on parallel computers, and none of the systems tested so far matches the scale of today's largest supercomputers.

Examples of real implementations of application-level checkpointing are *CLIP* [9], *Dome* [6], and *CCIFT* [8]. *CLIP* [9] is a checkpointing library for the Intel Paragon. Experiments were run on 128 processors with 32MB of memory each. The checkpoint interval was 15 minutes resulting in checkpoint sizes on the order of 10MB per node. The checkpoint system described in [6] implements fault tolerance within Dome's C++ objects. The evaluation was performed on 8 DEC Alpha workstations obtaining a checkpoint size of 3.3MB per node. The *CCIFT* (Cornell Compiler for Inserting Fault-Tolerance) [8] is a recently implemented method to provide fault tolerance at the application-level by the support of the compiler.

Ickp [20], *CoCheck* [21], *Diskless* [19], and *Starfish* [4] represent approaches to run-time library level checkpointing. *Ickp* [20] is a user-transparent checkpoint library for the Intel iPSC/860. It has been tested on a 32-node machine. Checkpoints were taken hourly obtaining checkpoint sizes for all of the test applications under 5MB per node. *CoCheck* [21] is a user-transparent checkpoint library for NOWs, built for process migration. It has been tested on an 8-node cluster of Sun SparcStation 2's and Sparc 10's connected by Ethernet. *Diskless* [19] is a user-transparent checkpoint library that uses the memory available on each node instead of saving the checkpoint to stable storage. Performance was tested on a cluster of 24 Sun Sparc-5 workstations, each with 96MB of physical memory and connected by a switched Ethernet network. Checkpoint size per node ranged from 4MB to 67MB using checkpoint interval ranging from 15 to 22 minutes. *Starfish* [3] is a user-transparent checkpoint library for clusters of workstations. The performance of this library was evaluated in [4] on a cluster of dual-processor Sun UltraSPARC-2's, each with 256MB of physical memory and connected by a 155-Mbps ATM network. Using a checkpoint interval of 10s, the checkpoint size per node ranged from 30MB to 120MB.

Checkpoint systems implemented at the operating system level are very rare. There are only a few works based on checkpointing for uniprocessor systems [7, 23], but to our knowledge, there are no works attempting to checkpoint parallel programs at the operating system level.

Finally, there are two recent simulation works on hardware-level checkpointing for Shared-Memory Multiprocessors, *Revive* [22] and *Safetynet* [25]. In *Revive* [22] the checkpoint is supported by modifying the hardware related to the directory controller of the machine in order to capture modifications of the address space of the application at the granularity of the cache line. The results presented were obtained by simulating a 16 node CC-NUMA multiprocessor. The checkpoint size ranges from 2.5MB up to 25MB when using checkpoints intervals of 10ms and 100ms, respectively. The overhead of this scheme is proportional to the L2 cache size. Despite using small caches (128KB), some applications may still exhibit overheads up to 22% due to their large working set. *Safetynet* [25] requires more hardware resources than *Revive*. The processor's caches must be modified, and it also requires an additional buffer to store the checkpointing data. Several short checkpoint intervals from 0.01ms to 1ms and buffers size from 256KB to 1MB have been analyzed.

8 Conclusions

This paper examined the feasibility of incremental checkpointing for scientific computing with an extensive experimental analysis.

Our results indicate that the implementation of automatic, frequent and user-transparent incremental checkpointing is a viable technique with current technology. Across a range of applications we found that the average bandwidth per process required to checkpoint is less than 100MB/s with a timeslice as small as one second. These figures are well below current technological limits in commodity clusters. They also suggest that automated techniques implemented below the application level may be sufficient. Also, these applications exhibit regular behavior that can be exploited to further optimize incremental checkpointing algorithms.

[2] www.hpcwire.com

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

Although our evaluation was performed on only 64 processors, we showed that these results can be generalized to much larger machines. In particular, the per process bandwidth requirements decrease slightly as processor count is increased and are sublinear in the application's memory footprint size. This provides an extra degree of robustness to our analysis.

Finally, by extrapolating the technological trends, we observed that future improvements in networking and storage will make incremental checkpointing even more effective.

Acknowledgments

Israel Hsu and Yuval Tamir provided substantial help in the development of the instrumentation library and in the initial phase of the experimental analysis.

This work was supported in part by the U.S. Department of Energy through the project LDRD-ER 2001034ER "Resource Utilization and Parallel Program Development with Buffered Coscheduling" and Los Alamos National Laboratory contract W-7405-ENG-36.

References

[1] D. Addison, J. Beecroft, D. Hewson, M. McLaren, and F. Petrini. Quadrics QsNet II: A network for Supercomputing Applications. In *Proceedings of the Hot Chips 14*, Aug. 18–20, 2003.

[2] N. R. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the Supercomputing, also IBM research report RC22570 (W0209-033)*, Nov. 16–22, 2002.

[3] A. Agbaria and R. Friedman. Starfish: fault-tolerant dynamic MPI Programs on Clusters of Workstations. In *Proceedings of the International Symposium on High Performance Distributed Computing*, Aug. 3–6, 1999.

[4] A. Agbaria and J. S. Plank. Design, Implementation, and Performance of Checkpointing in NetSolve. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 25–28, 2000.

[5] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Research Center, Dec. 1995.

[6] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneus Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, May 25, 1997.

[7] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems (TOCS)*, 7(1):1–24, Feb. 1989.

[8] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Principles and Practice of Parallel Programming*, June 11, 2003.

[9] Y. Chen, J. S. Plank, and K. Li. CLIP – A Checkpointing Tool for Message-Passing Parallel Programs. In *Proceedings of the Supercomputing*, Nov. 15–21, 1997.

[10] E. N. Elnozahy, L. Alvisi, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[11] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. Storm: Lightning-fast resource management. In *Proceedings of the Supercomputing*, November 2002.

[12] A. Geist and C. Engelmann. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors, 2002.

[13] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[14] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the Supercomputing*, Nov. 10–16, 2001.

[15] C. J. Li and W. K. Fuch. CATCH - Compiler assisted techniques for checkpointing. In *Proceedings of the Internationnal Symposium on Fault Tolerant Computing*, pages 74–81, Newcastle upon Tyne, UK, June 1990.

[16] E. Mainsah. Autonomic Computing: the Next Era of Computing. *IEEE Electronics Communication Engineering Journal*, 14(1):2–3, Feb. 2002.

[17] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Usenix Winter 1995 Technical Conference*, Jan. 16–20, 1995.

[18] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory Exclusion: Optimizing the Performance of Checkpointing Systems. *Software – Practice and Experience*, 29(2):125–142, Feb. 1999.

[19] J. S. Plank, Y. Kim, and J. J. Dongarra. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, Oct. 1998.

[20] J. S. Plank and K. Li. ickp — A Consistent Checkpointer for Multicomputers. *IEEE Parallel and Distributed Technologies*, 2(2):62–67, Summer 1994.

[21] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In *Proceedings of the IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 15–19, 1996.

[22] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, May 25–29, 2002.

[23] M. Russinovich and Z. Segall. Fault-tolerance for Off-the-shef Applications and Hardware. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 67–71, June 27–30, 1995.

[24] SCSI hard drive Seagate model Cheetah.

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture*, May 25–29, 2002.

[26] STREAM benchmark results. Available from `http://www.cs.virginia.edu/stream/standard/Bandwidth.html` .

[27] The ASCI Sweep3D Benchmark Code. Available from `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/`.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE