CLIP: A Checkpointing Tool for Message-Passing Parallel Programs

Yuqun Chen, James S. Plank, Kai Li

Abstract

Checkpointing is a useful technique for rollback recovery of parallel applications. While extensive research has been performed on checkpointing in parallel environments, there are few checkpointers available to application users on commercial parallel computers. This paper presents one such checkpointer: **CLIP**. **CLIP** is a user-level library that provides semi-transparent checkpointing for parallel programs on the Intel Paragon multicomputer. It is publicly available to Paragon users at no cost.

Conceptually, checkpointing a multicomputer is quite straightforward. However, when creating an actual tool for checkpointing a complex machine like the Paragon, many more issues arise that require careful design decisions to be made. Sometimes ease-of-use must be sacrificed for efficiency and/or correctness. This paper details what these decisions are, and how they were made in **CLIP**.

We also present performance data when checkpointing several long-running Paragon applications with **CLIP**. The bottom line is that a convenient, general-purpose checkpointing tool like **CLIP** can provide fault-tolerance on a massively parallel multicomputer like the Paragon with very good performance.

1 Introduction

Fault tolerance is very important for large-scale parallel computers. Typically, such machines have more hardware components than their workstation counterparts; its software is more complex and less tested. This results in a smaller mean time before failure. Since most applications for these computers are long-running and CPU-intensive, the ability to tolerate failures is of great importance.

Checkpointing is a useful technique for providing applications with *rollback recovery*. A checkpointing operation stores the execution state of a running program on stable storage. After a system restarts from a crash, the state of the application can be recovered from (or rolled back to) its most recent checkpoint and the program execution resumes as if the crash had never occurred.

This paper describes **CLIP**(**C**heckpointing **L**ibraries for the **I**ntel **P**aragon), a semi-transparent checkpointer for the Intel Paragon. **CLIP** can checkpoint programs written in either NX or MPI; both are *de facto* message-passing platforms for high-performance computing. To use **CLIP**, a user must link the **CLIP** library with his or her application program. Additionally, the user must place one or more subroutine calls in his or her code specifying when checkpoints can occur. When the code is executed, **CLIP** takes periodic checkpoints of the program's state to disk. If for some reason (hardware or software failure) the program is terminated prematurely, it may be restarted from the checkpoint file.

We call **CLIP** *semi-transparent* because the user must perform minor code modifications to define the checkpointing locations. This is more cumbersome than using a *totally transparent* checkpointer that can checkpoint any running application (but often requires operating system modifications). However, it is more transparent than many checkpointing tools [2, 11, 22], which require the user to rebuild the program state (i.e. the stack frames), the message state and the file system upon recovery. These tasks are handled automatically by **CLIP**.

The Paragon has been designed for high performance. Specifically, there is much custom hardware and specialized software whose purpose is to make network and file I/O extremely efficient. This complexity has many ramifications on the design of a checkpointing system. The bulk of this paper describes the design of **CLIP**, concentrating on five basic issues: (1) the degree of transparency, (2) checkpoint consistency, (3) protected message state, (4) efficient file I/O and (5) performance optimizations. For each issue, we detail the way in which **CLIP** inter-operates with the complex hardware and software of the Paragon to balance ease-of-use, correctness, and high performance.

The remainder of the paper assesses the performance of **CLIP** on several long-running applications on a large (128-node) Paragon installation. The conclusion that we draw is that **CLIP** is an effective tool for fault-tolerance on the Paragon, providing efficient checkpointing with a minimum of programmer effort.

2 Intel Paragon

The Intel Paragon is a MIMD-style distributed-memory multicomputer. A parallel application on Paragon consists of multiple instances (or processes) of the program, each running on a distinct *processing node*. The number of processing nodes in a Paragon can scale up to 1000. A Paragon node is composed of one or two *compute* processors and a *communication* processor. The processors share access to the node-private memory via a cache-coherent bus. The application process runs on the compute processor(s) in user mode. The communication processor runs the system-dependent layer of the message-passing protocol in protected kernel mode. The processors share memory so that communication may proceed in an extremely efficient manner without requiring the application to make costly system calls. This is described in greater detail in section 4.3.

The Paragon nodes are connected by a high-speed *mesh routing backplane* capable of 200 MB/sec node-to-node communication bandwidth. Processes of a program communicate with each other using either Intel's **NX** message-passing protocol [17] or **MPI** [15]. Both **NX** and **MPI** consist of three parts: user-level code linked with the application program, kernel-level code executing on the compute processors, and kernel-level code executing on the communication processor. This design enables users to use the raw power of the message-passing substrate while absorbing a minimum of software overhead.

A number of nodes in a Paragon are reserved as dedicated I/O nodes. They do not execute application processes, but instead cooperate to compose the *Parallel File System (PFS)*. Each has a hard disk attached to it. PFS is optimized for parallel I/O, performing striping across all the I/O nodes at a large granularity. Thus, large data reads and writes are efficient.

Thus, the Paragon can be configured to provide high-performance computing environment for application programmers. The challenge in writing a checkpointer for the Paragon is to enable fault-tolerance in a convenient manner for application users, while maintaining the high-performance of the computing environment.

3 CLIP Programming Interface

CLIP consists of two user-level libraries, `libNXckpt.a` and `libMPIckpt.a`, for **NX** and **MPI** programs respectively. Both libraries are semi-transparent, meaning the programmer must slightly modify his or her code to enable checkpointing. Specifically, the programmer must insert the procedure call `checkpoint_here()` at code locations where he or she desires checkpoints to be taken. Then, the programmer recompiles the modified modules, and re-links the program with the proper checkpointing library.

Upon instantiation, the user may set runtime parameters via command line arguments or via a control file. The startup portion of **CLIP** reads these parameters and acts accordingly. Examples of runtime parameters that the user may set are whether to checkpoint or recover from a checkpoint, the minimum interval between checkpoints, and which optimizations to employ.

While **CLIP** is less transparent than checkpointers that require no code modification [12, 13, 21, 24], it imposes minimal effort on the part of the user. There are several more cumbersome tasks that non-transparent checkpointers [1, 2, 11, 22] force the user to do that are totally automated in **CLIP**.

First and foremost is stack reconstruction. Non-transparent checkpointers provide the programmer with a convenient interface for saving/restoring critical data, but the user must rebuild the execution state of the program by hand. This means that if checkpointing occurs within one or more levels of procedure calls, the user must somehow re-instantiate those procedures upon recovery. Programming in such a way often results in awkward programs that are hard to read and maintain, and can be less efficient than they would be in the absence of checkpointing. **CLIP** saves the stack and registers as part of the checkpoint, meaning that the execution state is automatically restored upon recovery. This allows the programmer to insert `checkpoint_here()` calls anywhere in his or her code, regardless of the level of subroutine nesting.

Second is checkpoint consistency. As will be described in Section 4.2, the state of the network must be considered when taking a checkpoint of a parallel machine. If checkpoints are taken at the wrong times, it may be impossible to recover from a stored checkpoint. Non-transparent checkpointers deal with this issue by requiring the programmers to ensure that their checkpoints are consistent. In contrast, **CLIP** performs the task of assuring that checkpoints are consistent, thus relieving the programmer of that burden.

Third is the communication state of the processor. On the Paragon, a message may be buffered by a receiving node until the application asks for it. Thus, although it is not in the network, it is hidden from the application program until needed. If buffered messages are not stored with checkpoints, they will be lost on recovery. A non-transparent solution to dealing with communication state is to assume that there is none when checkpoints are being stored. In other words, the programmer must assure that all sent messages have been received by the application program before a checkpoint is taken. **CLIP** handles this problem by making sure that the buffers are stored as part of the checkpoint and then making them available on recovery.

Thus, although **CLIP** is not totally transparent, it achieves an ease-of-use that is far beyond non-transparent checkpointers.

4 The Design of CLIP

CLIP's ancestors are the checkpointers **libckpt** [19] and **ickp** [20]. **Libckpt** is a checkpointer for uniprocessor systems that will be discussed in detail in Section 4.5. **Ickp** is a checkpointer for the Intel iPSC/860 multicomputer. Because of **libckpt**'s implementation of effective checkpointing optimizations, and because the iPSC/860 is a much simpler machine than the Paragon, **libckpt** was chosen as the basic building block. **Ickp** was employed more as a "study guide" for dealing with consistency and message state.

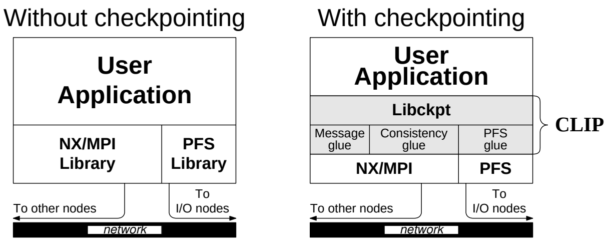

Figure 1: Structure of **CLIP**

The conceptual structure of **CLIP** is depicted in Figure 1. Without checkpointing, the user's application is linked with libraries for **NX** or **MPI**, and for PFS. These libraries handle communication and file I/O respectively. With checkpointing, the **CLIP** code acts as middle-ware between the user's application and the Paragon libraries. The main interface to checkpointing is the interface of **libckpt** [19]. However, there are three important issues on the Paragon that are not relevant for a uniprocessor checkpointer like **libckpt**. These are dealing with message state, processor consistency, and the parallel file system. **CLIP** integrates these issues with conceptual layers of glue between **libckpt** and the respective Paragon libraries.

The composite view of **CLIP**is thus as an implementation of **libckpt** with enhancements for working on the Paragon. The benefits of this approach are twofold. First, **libckpt**'s interface was designed to work hand-in-hand with several performance optimizations. These are described below in Section 4.5. By using **libckpt** as a building block for **CLIP**, we were able to incorporate a majority of **libckpt**'s optimization techniques. Second, application programmers who are familiar with **libckpt**'s approach will have an easy time adapting to **CLIP**.

In the remainder of this section, we discuss the main design decisions in the implementation of **CLIP**.

4.1 The degree of transparency

Ideally, checkpointing should be totally transparent to an application, implemented either in the operating system, or by an external program that can attach to a running executable and checkpoint its state without programmer or user intervention. However, the design of the Paragon precludes either of these approaches, leaving a checkpointing library as the most viable option.

Our initial desire was to make **CLIP** totally transparent, checkpoints being initiated by timer interrupts and signals. However, this approach was rejected because of its implications on correctness and performance. Instead, we opted for *semi-transparency*, where the programmer must place `checkpoint_here()` calls into his or her code. While this does place a burden on the programmer, it is a slight one compared to non-transparent approaches (discussed above in Section 3).

Below we detail the reasons why total transparency compromises both correctness and performance of the Paragon, and how the semi-transparency alleviates both of these problems[1].

First, message-passing on the Paragon consists of three parts – the user-level library, the compute processors' kernels, and the communication processor's kernel. All three contain data structures that reflect the current state of the network and communication system. With a totally transparent checkpointer, a checkpoint can occur anywhere in application code, including the message-passing or PFS library. This presents a problem because the kernel-level data structures cannot be checkpointed, whereas the user-level data structures can. This means if all three parts of the message-passing system rely upon one another, their states at the time of checkpointing cannot be saved and restored in their entirety.

The best solution to this problem is not to save any of the user-level data structures for the message system. Thus, upon recovery, the Paragon is assumed to be in a "clean" state: no messages are in transit or in the message buffers. If in fact the Paragon was not in a clean state upon checkpointing, we attempt to rebuild it on recovery by re-sending messages and creating a user-level message buffer. A subtle implication of this is that the checkpoint *cannot* occur in the middle of a message-sending procedure. This is because the message-sending procedure may rely on data structures that *will not be restored* when the system recovers from a checkpoint. By forcing the programmer to define the points at which the program can checkpoint, **CLIP** ensures that checkpointing never occurs in the middle of message-passing or PFS procedures.

Second, the **MPI** and PFS libraries both make heavy use of "system type" messages. These are messages that cannot be detected using the standard message-probing

[1] Implicit in this discussion is that we are unable to modify the kernel of the Paragon. This is because it is unreasonable to expect users to be able to install new kernel on the machines that they are using.

mechanism of **NX**. System type message cause problems for the checkpointer when it goes through its consistency checks because the checkpointer cannot make determinations about the state of the network using standard message-probing. This problem is fixed in **CLIP** by once again ensuring that a checkpoint cannot be taken in the middle of a Paragon library call.

Finally, taking transparent checkpoints means that blocking message-passing primitives (like `crecv()`, the synchronous message receiving subroutine in **NX**) should be interruptible. This is sometimes necessary to assure consistency of the checkpointed states. The only way to make blocking calls interruptible is to implement them with non-blocking calls, and "block" by polling until the operation is finished. Unfortunately, the synchronous message sending and receiving calls have been highly optimized in **NX** and **MPI**, and implementing them with their asynchronous counterparts imposes a significant amount of overhead in many cases. By employing semi-transparent checkpointing, the programmer assures that blocking calls do not have to be interrupted, and thus can utilize the higher performance synchronous messages passing primitives.

Thus, the decision to implement semi-transparent checkpointing simplifies the task of assuring both correctness and high performance. However, the burden on the programmer is small. In our experience, inserting the `checkpoint_here()` calls is straightforward even for complex applications that we did not write.

4.2 Checkpoint Consistency

When creating a checkpoint of a parallel program on a message-passing platform, *checkpoint consistency* is a fundamental problem. Simply stated, a checkpoint of a parallel program consists of a *processor state* and a *message state*. The processor state is composed of individual checkpoints for each processor in the system. The message state contains each message m such that m was sent by processor p_i before p_i's checkpoint, and received by p_j after p_j's checkpoint. There may be no messages m such that m is sent by processor p_i after p_i's checkpoint, and received by p_j before p_j's checkpoint. Otherwise, the checkpoint is inconsistent, and the checkpoint is unrecoverable.

Checkpoint consistency has been well-studied (see [5] for a thorough discussion of the area), and implementations have shown that a simple two-phased commit called "sync-and-stop" (SNS) exhibits performance on par with more complex algorithms [20]. Moreover, it is quite simple to implement. For these reasons, the SNS algorithm is implemented in the *consistency glue* module of **CLIP**.

When a `checkpoint_here()` call is reached and the minimum interval of checkpointing has passed, **CLIP** freezes the calling processor, and waits for all processors to call `checkpoint_here()`. At this point, any messages that have been sent prior to the `checkpoint_here()` call either have been received or are buffered in the kernel of the sender or receiver. Thus, the message state is well defined, and since all processors are frozen, there can be no messages sent after checkpointing that cause inconsistency.

Thus, when checkpointing, **CLIP** enforces a barrier synchronization. This barrier is implicit — the programmer need not perform a barrier in order to checkpoint. It simply occurs as a result of checkpointing. Since most programs for the Paragon follow a SPMD methodology, enforcing barrier synchronization should not be a problem. It is anticipated that most calls to `checkpoint_here()` will be made in places where synchronization occurs anyway, such as at the end of coarse-grained iterations, just before or after a natural communication phase of the program. In all of the applications that we instrumented for checkpointing, this was the case.

4.3 Protected message state

The high-level communication structure of a Paragon node is diagrammed in Figure 2. The application process on the compute processor(s) shares memory with the kernel of the communication processor. Specifically, they share a *system message buffer* for incoming messages, and a *post-page* as a communication buffer between the processors. By sharing memory, the communication processor can direct the communication to and from the network, and the application can access the results without absorbing any system call overhead.

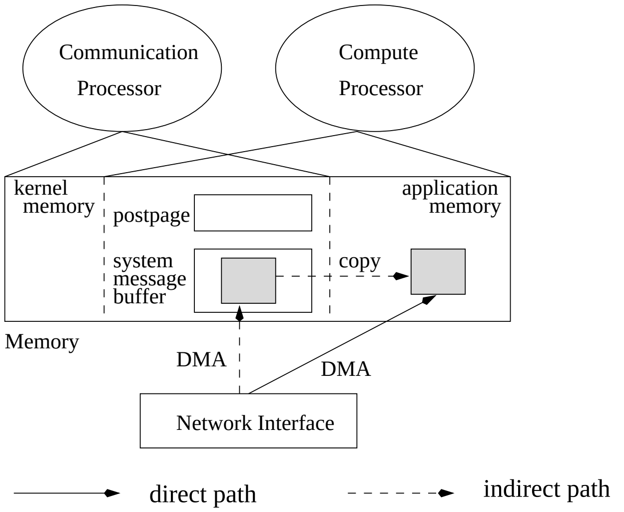

Figure 2: Communication structure of a Paragon node

There are two paths that a message can take when it arrives at its destination node: a *direct* path and an *indirect* path. If the user has already posted a receive for the message, then the communication processor instructs the network interface to perform DMA directly to the user's receive buffer. This, the direct path, is the fastest way for the message to arrive at the proper destination. If the user has not posted a receive for the message, then the the communication processor instructs the network interface to perform DMA into the system message buffer. When the user subsequently goes to receive the message, it is copied from the system buffer to the user's buffer by the compute processor. Since the system buffer is shared, this activity involves no system call overhead.

The message-sending primitives on the Paragon may be either synchronous (blocking), or asynchronous (non-blocking). When an asynchronous send or receive call is made, the communication processor ensures that the action is performed, and sets flags in the post-page to store the result. To complete the asynchronous operation, the application checks the post-page, and clears the flags. Again the fact that communication processor shares memory with the application allows all of this to occur without system call overhead.

When checkpointing an application, the buffered messages and the state of asynchronous communication must be stored with the checkpoint. A major design decision of **CLIP** is how this should be done. A natural and efficient way to store this information would be to simply checkpoint the post-page and system message buffer. However, this solution requires the communication processor to be aware of checkpointing so that it does not modify the shared memory while a checkpoint is being stored. Moreover, upon recovery, the internal data structures of the communication processor would have to be rebuilt to reflect the communication state of the node. Since one of our goals was not to modify the kernel code of the Paragon, we had to reject this solution.

Instead, when a checkpoint is being taken, **CLIP** "probes" the message buffer using an **NX** primitive, and extracts messages from the system buffer by calling the synchronous receive operation. These messages are stored in non-shared memory within the *Message Glue* module of **CLIP**, which gets checkpointed with the application. The shared state between the communication and compute processors is *not* stored as part of the checkpoint. Thus, whenever the application makes a message receive call, the **CLIP** buffer is first scanned. If the message is found in **CLIP**'s buffer, then it is copied to the application's buffer and the receive is marked as complete. Otherwise, the appropriate Paragon receive primitive is called.

This is the same solution to message buffering as used by **ickp** on the Intel iPSC/860 [20]. Note that it adds an extra level of message copying to the indirect path of message receiving. However, the faster, direct path is unaltered, because the semi-transparent nature of **CLIP** combined with the SNS checkpointing algorithm insure that the DMA from the network interface can not be interrupted by checkpointing.

Asynchronous operations are dealt with in a similar manner. Before checkpointing, for each outstanding asynchronous operation, **CLIP** tests the operation for completion. If the operation is complete, then nothing need be done except store the result for when the application tests for completion. If the operation is not complete, then the parameters of the original operation are stored in the checkpoint so that it may be re-executed upon recovery. Again, semi-transparency and the SNS algorithm assure that the state of asynchronous operations will not change during checkpointing, insuring the integrity of the tests for completion in **CLIP**.

There are a few more details concerning the Message Glue module, such as dealing with certain **MPI** calls and maintaining the correctness of the core communication state upon recovery. However, these details are omitted for brevity.

4.4 Efficient file I/O

Earlier experimental research in checkpointing parallel and uniprocessor programs have pointed out that the major source of checkpointing overhead comes from committing the checkpoints to the stable storage. The PFS Glue in **CLIP** attempts to reduce this overhead in two ways:

- smart layout of the checkpoint file,
- efficient use of PFS.

Because **CLIP** implicitly excludes some runtime data (e.g. shared communication state) and the application may exclude some application data from checkpointing, **CLIP** often has to checkpoint a fragmented data space. Each contiguous segment of the data needs a small header to identify its size and location. The original version of **libckpt** interleaves the segment headers with the contents of the segment in the checkpoint file. On the Paragon, this results in frequent small write requests to the I/O nodes which are optimized for handling large-size I/O requests. Instead, **CLIP** adopts a different layout of the checkpoint which coalesces all segment headers in a large buffer and writes it out to the disk in one operation.

Figure 3 depicts the file layout of checkpoint data for a single process. The checkpoint header, whose size is aligned to the file system block size, contains general information about the checkpointed program followed by a sequence of segment headers. The actual contents of the segments are stored contiguously after the checkpoint header. The stack is stored as the last segment, and the processor context, including the program counter and the register file, is stored in a global data structure in the heap.

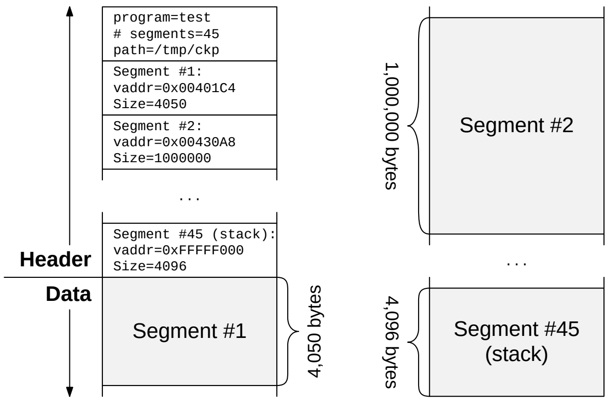

Figure 3: Layout of checkpoint file

CLIP allows checkpoints to be stored in Unix files or in PFS files. Obviously, PFS allows **CLIP** to achieve better performance. When each process writes its checkpoint data to a PFS file, PFS automatically *stripes* the data at a fixed striping unit size. For optimal performance, the striping unit size should be a multiple of the file block size. This is 64 kilobytes each on Paragon to optimize large I/O requests. For peak performance, **CLIP** attempts to output checkpoint data in multiples of optimal PFS striping unit, using a large internal buffer to gather segments smaller than the striping unit.

Opening a PFS file on the Paragon is an expensive operation, because it involves Mach IPC communication with all the I/O nodes in the parallel file system. To avoid the high cost of opening multiple checkpoint files, such as would happen if every node opened its own checkpoint file, **CLIP** opens just two large PFS files, one for the current checkpoint, and one for the previous checkpoint. This is made possible by the **M_ASYNC** access mode, which lets a collection of compute nodes read and write different portions of a large file simultaneously without interprocess coordination. The checkpoint file is partitioned into a number of subregions (or *subfiles*), each of which stores the checkpoint data for a distinct process. The checkpoint headers from all processes are grouped together at the beginning of the checkpoint file. In addition to high throughput for writing checkpoints to the disks, the subfile organization also simplifies the management of checkpoint files.

4.5 Performance Optimizations

One of the main strengths of **libckpt** as a checkpointing library is its implementation of performance optimizations. Since **CLIP** is built on top of **libckpt**, it too benefits from several performance optimizations.

First is *incremental* checkpointing [9], which employs virtual memory primitives so that only pages that have been modified since the previous checkpoint are written to the checkpoint file. If a program contains a significant amount of *read-only* data from checkpoint to checkpoint, incremental checkpointing can improve the performance of checkpointing.

Second is *programmer-directed memory exclusion* [19]. Since the programmer knows where checkpoints will occur, he or she can identify parts of the program that may not need to be checkpointed. For example, at the end of an iteration or subcomputation, there may be temporary arrays whose values are not used any further in the computation. **Libckpt** provides two procedures `exclude_bytes()` and `include_bytes()` that enable programmers to specify when not to include regions of memory in a checkpoint, and when to re-include them if necessary. For many programs, a few memory exclusion calls can yield huge savings in the size and overhead of checkpointing [19]. Memory exclusion as defined by **libckpt** is included as a part of **CLIP**.

Third is *implicit memory exclusion*. If the programmer is using a memory allocator (such as `malloc()` in C), there may be large blocks of memory on the allocator's free list that do not need to be checkpointed. In **CLIP**, the standard Paragon allocator has been modified to exclude any memory on the free list. This simple tweak can be responsible for large savings in checkpoint size and overhead because the Paragon's allocator uses the *buddy system*. In the buddy system, all memory requests are rounded up to the nearest power of two. This leads to a fast allocation algorithm that returns page-aligned addresses for large requests. The problem with the buddy system is that if the large requests are not powers of two, then much of the allocated memory is unused. Performance-wise, this is not a big problem because the Paragon only maps a page to physical memory if the page is actually used. However, if the checkpointer is unaware of this fact, it can severely degrade performance because it will attempt to checkpoint these unused pages, which results in their being mapped and then written to disk. In extreme cases, this could cause a program that runs correctly in the absence of checkpointing to run out of memory when checkpointing is enabled. **CLIP** solves this problem by ensuring that both free memory and unused allocated memory are excluded from checkpoint files.

The only optimization implemented by **libckpt** but

Name	Message Passing Library	Language	Running Time (sec)	(mm:ss)	Address Space Size (MB) (per node)	(total)	Minimum Checkpoint Size (MB)	Checkpoint Interval (min)	Failure-free Overhead (sec/ckp)	(%)
CFD	**MPI**	Fortran	4004	66:44	10.3	1320	1303	15	27.5	3.7
PRISM	**NX**	C	4684	78:04	22.6	2895	421	18	22.0	1.9
RENDER	**NX**	C	3360	56:00	14.3	1849	137	14	9.3	1.1
PSTSWM	**NX**	Fortran	3215	53:60	9.0	1153	509	14	9.3	1.2

Table 1: Basic performance for each application

omitted from **CLIP** is the copy-on-write optimization, because it is rare for a Paragon application have the free memory necessary to buffer checkpoints and write them asynchronously.

5 Performance

This section presents the performance of **CLIP** when checkpointing and recovering several large-scale parallel applications. The purpose of this section is to assess the performance of **CLIP**, a real checkpointer for a commercial machine, on several real-world applications so that the efficacy of **CLIP** can be gauged. This section is *not* meant to evaluate different performance optimizations, since such work has been performed many times in the past (e.g. [4, 6, 14, 19]).

5.1 Applications

We chose several **NX** and **MPI** applications to test the performance of **CLIP**. These applications are written in C or Fortran, and have been chosen from various sources. They are either real applications used to solve large engineering problems or parallel benchmark programs that closely mimic real and complex parallel programs. Table 1 summarizes the applications and their basic performance under **CLIP**. We briefly describe them below.

- **CFD** is the "LU" program from the NASA Parallel Benchmark Suite NPB2.1[8]. It is a three dimensional computational fluid dynamics simulation.
- **PRISM** is a three-dimensional Navier-Stokes turbulence simulation program written at Caltech [10].
- **RENDER** is a program from JPL that combines satellite imagery with terrain elevation to produce three-dimensional perspective views of Mars and Venus surfaces [16].
- **PSTSWM** is a benchmark program from Oak Ridge National Laboratory[25] that performs the core part of many weather modeling programs.

All of the programs besides **RENDER** work in globally synchronized iterations. Thus it is straightforward to insert `checkpoint_here()` calls at the end of iterations, and then set a runtime parameter to force checkpointing at a desired interval. **RENDER** works in a master/slave fashion, where one master node allocates tasks to the other slave nodes. As such, there is no natural synchronization point to insert `checkpoint_here()` calls directly. Nonetheless, it was trivial for us to add a new message type to the program that the master process broadcasts to all slaves when it decides on an appropriate time to checkpoint. Upon receiving the checkpoint message, all processes immediately call the `checkpoint_here()` routine to initiate checkpointing.

5.2 Experiment Methodology

We used a 512-node Intel Paragon at Caltech. Each node contains one compute processor and one communication processor, plus 32 MB of memory. The PFS subsystem has 64 I/O nodes, each of which is attached to a disk drive and contains 64 MB of memory. The Caltech Paragon is a production-level system that experiences heavy use daily. As such, we were able to run the applications on only 128 nodes (for **RENDER**, we use 128 slaves and one master node). The heavy use of the Caltech Paragon introduces a small amount of fluctuation into our performance measurements, because other applications can use the parallel file system at the same time as our program. To combat this, we ran each test three or more times to get an average measurement.

The data set for each application is chosen so that the application runs in a normal manner for approximately one hour. This lets us approximate long-running applications without artificially increasing loop iterations. For each application we produced two versions of executables: a *pure* version with the original compilation configuration and a *checkpointed* version that has `checkpoint_here()` inserted and is linked with **CLIP**. We set the checkpointing interval to be 15 minutes, which results in 4 checkpoints for each run.

Where possible, we performed programmer-directed memory exclusion, and combined it with implicit memory exclusion and incremental checkpointing.

5.3 Checkpoint Sizes

Figure 4 shows the average size of the checkpoint files for each application when using (1) no optimization, (2) implicit memory exclusion, (3) implicit and programmer-directed memory exclusion, and (4) all of the above plus incremental checkpointing. Note that case (1) is equivalent to the "Address Space Size" column in Table 1, and that case (4) is equivalent to the "Minimum Checkpoint Size" column.

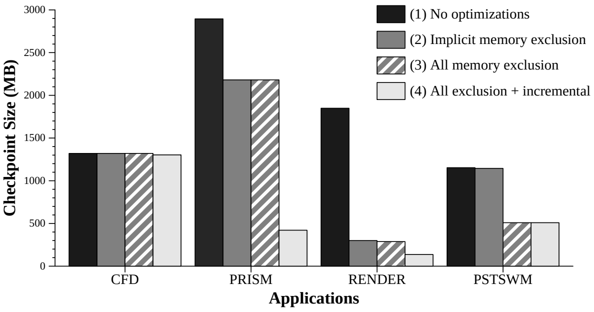

Figure 4: Checkpoint Sizes

As other optimization studies have shown, the degree of space optimization varies from program to program. Some, such as **CFD**, can not be optimized by these techniques. Incremental checkpointing shrinks the checkpoints of **PRISM** from a size of 2.9 GB to 421 MB. Implicit memory exclusion reduces the size of **RENDER** checkpoints from 1.8 GB to 137 MB, and programmer-directed memory exclusion cuts the size of **PSTSWM** checkpoints roughly in half.

For each application, the results that follow use the combination of optimization techniques that performs the best for that application.

5.4 Failure-free Overhead

The *failure-free overhead* introduced by a checkpointer is the additional time it takes a checkpointed application to run from start to completion without encountering any failures. Failure-free overhead is a comprehensive measure of checkpointing latency (overhead in the protocol and efficiency at outputting checkpoints) as well as overhead the checkpointer introduces into message passing and other parts of the program.

We measure the failure-free overhead for each application by subtracting the running time of the pure version of the program from the running time of the checkpointed version. This overhead is tabulated in the last two columns of Table 1. The overhead per checkpoint is the total overhead divided by the number of checkpoints (four), and the percent overhead is the total overhead divided by the running time of the pure version of the program.

The major conclusion to draw from this data is that the overhead of checkpointing is extremely small considering the size of the programs. Most checkpointers target 10 percent overhead as their desired upper limit, and **CLIP**'s performance is far better than that. This is mainly the result of three factors: the aggregate bandwidth of disk writing with PFS, the design of **CLIP** that gets maximal performance from PFS, and the checkpointing optimizations.

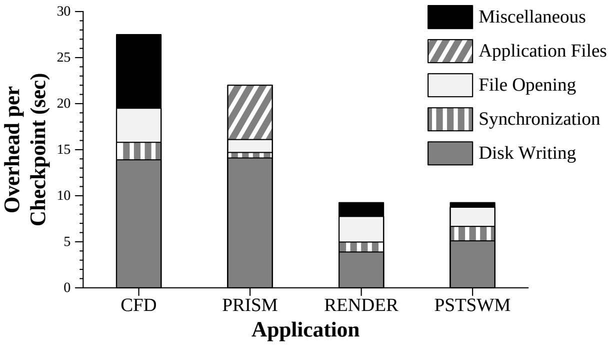

Figure 5: Composition of checkpoint overhead

In Figure 5, we break down the average overhead per checkpoint into its constituent parts. These are as follows:

- *Disk Writing*: This is the time spent performing actual disk writes. Typically, this is the major portion of overhead, and Figure 5 shows this to be the case.

- *Synchronization*: This is the time spent ensuring that all of the nodes are in a consistent state before checkpointing. As shown previously [7, 20], this is not a significant portion of checkpointing overhead.

- *File Opening*: As stated above, opening large files in PFS is a very expensive operation. **CLIP** opens two checkpoint files upon instantiation, and then uses those for all checkpointing. In Figure 5, this overhead is amortized over four checkpoints. As more checkpoints are taken, this portion of overhead will become negligible.

- *Application Files*: If the application has opened files, **CLIP** flushes their state to disk and checkpoints the file pointers. Like file opening, this is an expensive operation, and is the cause of a significant portion of the overhead in **PRISM**. It is not a factor in the other applications.

- *Miscellaneous Overhead*: This accounts for the remaining overhead, which includes managing the memory exclusion, catching page faults for incremental checkpointing, extracting messages from the system buffer, etc.

While we expect disk writing to compose the most significant portion of checkpointing, we note that other factors, most notably the overhead of opening files are significant as well. This is an important observation for the fact that most checkpointing research focuses on either minimizing the number of disk writes, or minimizing the messages in the consistency protocol.

Name	Recovery Time (sec)
CFD	36.0
PRISM	33.4
RENDER	22.5
PSTSWM	19.0

Table 2: Performance of recovery

5.5 Recovery

Table 2 shows the time that it takes each application to recover from a checkpoint and to begin executing the application again. Note that recovery times are larger than the average checkpoint times for each application. This is mainly for two reasons. First, while the overhead of opening files is amortized over all checkpoints, it is included in its entirety in the recovery time. Second, if a program contains a lot of read-only data, then with incremental checkpointing the first checkpoint is large, and successive checkpoints are small, improving the average checkpoint size and overhead. However, recovery involves restoring all data (read-only and not read-only), and therefore can exhibit much more overhead than during checkpointing.

Since recovery is a less frequent operation than checkpointing, the higher recovery times are permissible. It should be noted that in no case is the recovery time greater than 10 percent of the application's running time.

Checkpointing Method	Total Overhead (sec)	Checkpoint Size (MB)	Percent Overhead
Native	43	77	0.9
CLIP	89	421	1.9

Table 3: Native checkpointing vs. **CLIP** in **PRISM**

5.6 Comparison with Non-transparent Checkpointing

The **PRISM** application has its own checkpointing and restart mechanism coded in the program which can be enabled or disabled via a runtime option. This non-transparent checkpointing mechanism saves critical application data into a file at certain loop iterations. To recover, **PRISM** is invoked with a different runtime option which reads the critical data and restarts from the middle of the computation.

It is interesting to compare the performance of **CLIP** with PRISM's non-transparent checkpointing mechanism. The relevant data is in Table 3. The native checkpointer's files are smaller by a factor of 5.5, but the overhead is only a factor of 2 faster. **CLIP**'s better performance per megabyte is due to its optimized use of PFS. Considering the fact that checkpointing with **CLIP** is far easier than coding checkpointing and recovery explicitly into the application, the extra 1% overhead that **CLIP** adds to checkpointing seems minor indeed.

6 Conclusion

This paper has described **CLIP**, a semi-transparent tool for checkpointing parallel programs on the Intel Paragon. **CLIP** is significant because it is a real tool for a real machine that is available for free on the Internet. There have been checkpointing tools previously released for uniprocessors [19, 24], distributed systems [3, 23], and even multicomputers [20], but none for a machine with the size, performance and complexity of the Paragon. Although **CLIP** runs only on Intel Paragon, its design techniques may apply to other multicomputer platforms.

Writing a checkpointing tool for a machine such as the Paragon is more difficult than taking a simple core dump. There are many tradeoffs in performance, functionality and ease-of-use that must be addressed in the design stages. This paper details several of these issues and should be valuable to anyone who designs and implements tools for complex machines such as the Paragon.

The performance of **CLIP** is excellent, mainly due to the efficient distributed I/O of PFS. The bottom line is that with a tool like **CLIP**, a large application for a massively parallel machine such as the Paragon can be instrumented with fault-tolerance with little burden on the programmer or on the performance of the application.

The memory exclusion technique (*both programmer-directed and implicit exclusion)* can be effective in reducing the checkpointing overhead. In our experiments, memory exclusion reduced checkpoint size by between 25% and 90% for three out of four applications. Furthermore, compilers can also take advantage of the memory exclusion interface to *automatically* exclude certain application data from checkpoints [18].

Although **CLIP** is an effective semi-transparent checkpointer, there are still open design questions. For example, there are programs for which `checkpoint_here()` is not very convenient to use. It remains an open problem to design and implement a transparent checkpointer with memory exclusion interface. Even if there is such a design, it is unclear how to implement it efficiently. We have only limited application experience with **CLIP**. Although memory exclusion API is quite convenient for uniprocessor applications, it is unclear whether such an API is easy to use with parallel applications. We would like to hear feedbacks from a wide range of users. **CLIP** is freely available on the Internet at:

`http://www.cs.princeton.edu/sio/CLIP.html.`

7 Acknowledgment

This project is supported in part by the Scalable I/O project under the DARPA grant DABT63-94-C-0049 and by National Science Foundation under grant CCR-9409496. We would like to thank the members of the Scalable I/O Initiative for providing us with the applications used in this study. Ronald D. Henderson at Caltech and Patrick Worley at Oak Ridge National Laboratory gave us insightful suggestions for instrumenting their programs. We also would like thank Sharon Brunett and Heidi Lorenz-Wirzba at Caltech for helping us with running our experiments on the Caltech Paragon.

References

[1] A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A novel software system for fault-tolerant parallel processing on distributed platforms. In *4th IEEE International Symposium on High Performance Distributed Computing*, August 1995.

[2] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. Technical Report CMU-CS-96-157, Carnegie Mellon University, to appear in *Journal of Parallel and Distributed Computing*, August 1996.

[3] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *3rd Annual PVM Users' Group Meeting*, Pittsburgh, PA, May 1995.

[4] T. Chiueh and P. Deng. Efficient checkpoint mechanisms for massively parallel machines. In *26th International Symposium on Fault-Tolerant Computing*, Sendai, June 1996.

[5] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, October 1996.

[6] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[7] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307, Austin, TX, June 1994.

[8] David Bailey et al. The nas parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[9] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[10] Ronald D. Henderson and George Em Karniadakis. Unstructured spectral element methods for simulation of turbulent flows. *Journal of Computation Physics*, 122(2):191–217, December 1995.

[11] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *23rd International Symposium on Fault-Tolerant Computing*, pages 2–9, July 1993.

[12] B. A. Kingsbury and J. T. Kline. Job and process recovery in a UNIX-based operating system. In *Usenix Winter 1989 Technical Conference*, pages 355–364, San Diego, CA, January 1989.

[13] C. R. Landau. The checkpoint mechanism in keykos. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, September 1992.

[14] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1.0 edition, May 1994.

[16] A. Chien P. Crandall, R. Aydt and D. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of SuperComputing 95*, 1995.

[17] Paul Pierce. The Paragon implementation of the NX message passing interface. In *Proceedings of Scalable High-Performance Computing Conference (SHPCC) 94*, 1994.

[18] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.

[19] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under unix. In *Usenix Winter 1995 Technical Conference*, pages 213–223, January 1995.

[20] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[21] M. Russinovich and Z. Segall. Fault-tolerance for off-the-shelf applications and hardware. In *25th International Symposium on Fault-Tolerant Computing*, pages 67–71, Pasadena, CA, June 1995.

[22] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In *Scalable High Performance Computing Conference*, pages 694–701, Knoxville, TN, May 1994.

[23] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium*, April 1996.

[24] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobb's Journal*, #227:40–48, February 1995.

[25] P.H. Worley and I.T. Foster. Parallel spectral transform shallow water model: A runtime-tunable parallel benchmark code. In *Proceedings of Scalable High-Performance Computing Conference (SHPCC) 94*, 1994.