Application Level Fault Tolerance in Heterogeneous Networks of Workstations

Adam Beguelin Erik Seligman Peter Stephan

August 1996
CMU-CS-96-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A condensed version of this paper will appear in a special issue of the *Journal of Parallel and Distributed Computing* on Workstation Clusters and Network-based Computing, June, 1997.

Abstract

We have explored methods for checkpointing and restarting processes within the Distributed object migration environment (Dome), a C++ library of data parallel objects that are automatically distributed over heterogeneous networks of workstations (NOWs). System level checkpointing methods, although transparent to the user, were rejected because they lack support for heterogeneity. We have implemented application level checkpointing which places the checkpoint and restart mechanisms within Dome's C++ objects. Application level checkpointing has been implemented with a library-based technique for the programmer and a more transparent preprocessor-based technique. Dome's implementation of checkpointing successfully checkpoints and restarts processes on different numbers of machines and different architectures. Results from executing Dome programs across a NOW with realistic failure rates have been experimentally determined and are compared with theoretical results. The overhead of checkpointing is found to be low, while providing substantial decreases in expected runtime on realistic systems.

This research was sponsored by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives and by the Advanced Research Projects Agency under contract number DABT63-93-C-0054.

Keywords: Heterogeneous parallel programming, architecture independent checkpoint and restart, application level fault tolerance, Dome, PVM

1 Introduction

Using clusters of workstations to solve large scientific problems is a current focus in the high performance computing field. While this method offers many advantages, it also creates some problems. Handling failures on one or more of the nodes in the cluster becomes an important concern. As the number of workstations in a cluster increases, the chance that one of them will fail during a particular computation increases exponentially. For example, on a workstation with a mean time between failures of 16 days, a one day computation may have 94% chance of completing successfully, while on a cluster of ten such machines, there is only a 54% ($.94^{10}$) chance that a one day computation will complete before a failure occurs. Thus, it is vital that some kind of fault tolerance mechanism be incorporated into any system designed for extended execution on a workstation cluster. This paper discusses the implementation of fault tolerance mechanisms at various levels of programming abstraction and specifically describes the results of the initial implementation of some of these methods which have been developed for use with the Distributed object migration environment (Dome) [1, 2].

Dome is a system that is designed to provide application programmers a simple and intuitive interface for parallel programming in a heterogeneous environment. It is implemented as a library of C++ classes and uses PVM [10, 9] for its process control and communication. When an object of one of these classes is instantiated, it is automatically partitioned and apportioned among the nodes of the workstation cluster. Dome uses a single program multiple data (SPMD) model to perform parallelization of the program. In the SPMD model the user program is replicated on each machine in the cluster, and each copy of the program, executing in parallel, performs its computations on a subset of the data in each Dome object. For a more complete discussion of Dome, see [1].

A fault tolerance package for use with Dome can be implemented at various levels of programming abstraction. At the application level the programmer can call a set of C++ methods to checkpoint a program's data structures and to restart that program from the checkpointed data. This method provides a fault tolerance package which is highly portable since it uses no machine-dependent constructs in creating a checkpoint. The application level method is not transparent to the user, however, as it requires the application programmer to insert the calls to the checkpoint and restart mechanism. A refinement of the application level method uses a preprocessor to insert most of the checkpointing calls automatically. The use of a preprocessor offers the same advantages of portability while helping to reduce the work required of the application programmer. System level checkpointing methods periodically save the program's memory image upon interrupt. These methods are very simple to use, requiring no additional work of the programmer, and the program can be restored easily from the saved checkpoints. However, neither the implementation of the system level fault tolerance packages nor the checkpoint files that are produced are generally portable to other platforms. Furthermore, determining a consistent state for the program is a difficult issue since communication may be in progress between the distributed processes at checkpoint time. With the application level methods the knowledge of the program structure alleviates this problem.

While they tend to require more work from the user, the application level fault tolerance methods mesh very well with the Dome system since both allow for easy portability to any system that supports PVM and C++. Furthermore, since Dome applications execute in a heterogeneous environment, it is vital that checkpoints created on one architecture are usable on others. Thus, this work concentrates on the design, development, and implementation of application level checkpointing features for Dome.

This paper describes an implementation of an application level checkpoint and restart package for use with Dome, and benchmarks that have been collected using this package. Both a library-based checkpoint mechanism and a preprocessor-based mechanism are presented as well as a failure daemon to facilitate program restart in the presence of failures. Timings of a molecular dynamics application which uses Dome were collected. These timings indicate that even when checkpointing is performed very frequently, the overhead is low enough to provide a good expected runtime for an application.

2 Checkpointing in Dome

Dome programs are written using a library of distributed objects. These objects may be fragmented and the portions of each object distributed over a number of multi-user heterogeneous computers. Dome controls the distribution and layout of the objects and may alter that distribution periodically for load balancing purposes. The distribution and load balancing is transparent to the programmer. When programming in Dome, the parallelism is implicit in the operations performed on objects. For instance, the Dome distributed vector (`dVector`) class overloads the addition operator allowing for the parallel addition of two `dVectors` by simply writing $a + b$, where a and b are

dVectors. By design, Dome programs use the single program multiple data (SPMD) model in parallelizing the work. All of the communication among tasks is done within the implementation of Dome objects; therefore, the user does not perform any explicit message passing. This design has several significant advantages with respect to a fault tolerance mechanism. Because Dome controls the distribution of a program's objects, it can also checkpoint this data in an architecture independent form, allowing the program to restart on a different architecture. Because communications operations are embedded within the objects, the user is free to place checkpoints into the program with few restrictions. Since Dome controls the mapping of data to processors, most Dome programs can be mapped to any number of processors. This allows Dome's restart mechanism the flexibility of mapping a Dome program to a different number of processors than were originally in use at the time of the checkpoint. Finally, the SPMD structure of Dome programs allows a checkpoint to be taken without global synchronization. These independently generated checkpoints can then be reconstructed into a global checkpoint whenever a restart is necessary.

The model for this fault tolerance package assumes that all tasks will be fail-stop. That is, either the program runs to completion and produces the correct results, or the run terminates prematurely and informs the user of this fact. Currently the system detects the failure and attempts to restart the application from the most recent checkpoint. Failure notification is provided by the underlying system, PVM in this case. (See Section 2.5 for more details.) This work is not intended to address the question of program errors or arbitrary failure modes which may corrupt the program results without the user's knowledge. This model describes a fault tolerance package that periodically saves the current state of the distributed program to one or more checkpoint files and allows the program to be restarted from the most recent checkpoint after a failure. Once restarted, the program should proceed normally from the position of the last checkpoint.

2.1 Application State

There are several pieces of information which must be stored in order to restart a program successfully after failure. In general, this consists of the user memory, stack, registers, messages in transit, and relevant operating system variables. At checkpoint time, though, it is only necessary to save sufficient information to restart the program successfully from the checkpoint.

The application level checkpoint and restart mechanism operates entirely in C++ code and, therefore, has no access to internal machine information such as register and stack values. In order to provide a portable package, it is important to restrict dependence on machine-specific information in the fault tolerance mechanism. The relevant state necessary to restart applications using an application level model consists of:

- **The program counter.** The fault tolerance package must determine exactly where in the program the last checkpoint occurred. This can be difficult in an application level checkpointing mechanism. The problem is simplified in our application level model by guaranteeing that checkpoints will only occur at user-inserted calls to a `dome_checkpoint()` method.

- **The stack.** A set of procedure calls will be active at the time of checkpoint. The stack must be restored so that the application can properly return from each procedure call after restart as if there had not been a failure. This information is difficult to save without looking into the values on the internal stack, and, once saved, reconstructing it upon restart is not easy. The checkpoint preprocessor described in Section 2.3 addresses this problem by inserting a set of procedure calls which keep track of the call stack so that it can be stored at the time of a checkpoint.

- **A subset of program variables**. Since many program variables are temporary, their values are only relevant during certain parts of the program execution. Only the variables required to restore the program to the position of a given checkpoint need to be saved. For Dome programs this is defined as the set of all Dome objects. It is the responsibility of the programmer to ensure that the relevant data is encapsulated in Dome objects. For base types this is as simple as a slight change in variable declarations. For example, `int` becomes `dScalar<int>`, a templated Dome class that simply implements checkpoint and restart methods for scalars. The templated nature of this class facilitates its extension to other user defined data types. Checkpointing pointers and user-allocated memory, however, presents a bit of a challenge. These constructs could be addressed by a "Dome pointer" class whose allocation method would store the type and size of the allocated memory. Thus, the memory block can be saved during checkpoint, and re-allocated and loaded upon restart. For allocated memory which contains pointers, such as a linked list, this scheme will not work. New Dome objects would need to be created specifically for linked lists and similar structures. The object oriented structure of the Dome library makes this task tractable. It should also be noted that the C++ pass-by-reference mechanism replaces many of the uses of pointers in C. Thus, while these checkpointing methods for pointers and user-allocated memory are not fully general, they should be useful in most practical cases.

- **Communication state.** In a general parallel program there may be messages in transit while a checkpoint is being written. In Dome, however, communication only occurs within calls to methods in the Dome library. The user does not make explicit communication calls. Thus, since the application level checkpoints are taken between calls to Dome methods, it is guaranteed that no communication state will need to be saved. This fact, in conjunction with the SPMD structure of Dome programs, allows for synchronization-free checkpointing. The current implementation synchronizes to create a single checkpoint file. A synchronization-free version would merge the independently created checkpoint files upon restart.

- **I/O state.** If the program performs any I/O operations the fault tolerance mechanism must allow for proper restart of those operations. This can be an extremely difficult task depending on the variety or complexity of I/O operations being performed. The current model addresses only the issue of file I/O operations. In most cases files are simply read from or written to as a continuous stream. Therefore, our checkpoint method saves the file pointer at the time of checkpointing and opens the file at that location upon restart. File operations that do not fit this description present a greater challenge. For these operations a logging mechanism is needed so that changes to files can be undone. This case is not currently implemented in the Dome system.

2.2 Library-Based Application Level Fault Tolerance

In this section we describe the library-based application level checkpointing which is a foundation for the more user friendly preprocessor-based checkpointing subsequently described in Section 2.3.

In library-based application level checkpoint and restart the fault tolerance package must operate entirely in C++ code without using machine-specific details, looking at the internal machine state, or requiring any special compiler. To do this successfully while minimizing the amount of code the user has to write, some restrictions must be placed on the programming model. It is likely, however, that a wide variety of scientific applications will adapt well to this model. A less restrictive application level fault tolerance model is described in Section 2.3.

The model that a program must follow in order to use the library-based application level checkpointing package is illustrated in Figure 1. The program first initializes the Dome environment. By starting the program with the appropriate command-line arguments, the user may indicate that the program is to start from the beginning or restart from a saved checkpoint. Next the Dome variables, here consisting of distributed vectors (`dVector`) and distributed scalars (`dScalar`), are declared. `dVectors` are automatically distributed by the Dome library while `dScalars` are ordinary variables that are replicated in each process. All Dome variables are registered with the Dome environment upon declaration. The Dome environment maintains pointers to those variables and at program checkpoint time they are written to the checkpoint file. Therefore, all program variables which need to be saved to restart the application successfully must be declared as Dome variables. Dome uses templated objects to allow the application programmer considerable freedom in declaring Dome variables of different types. The overhead of using `dScalars` over normal C++ variables has been measured at only about 1-2% in the Dome molecular dynamics application. If the application is restarting, a Dome variable declaration causes the value of that variable to be read in from the checkpoint file. Following the variable declaration section, the program should execute its initialization code. If the program is restarting from a checkpoint file, the values of the program variables will have already been restored. In this case the initialization code is skipped. The programmer determines whether the program is restarting by making a call to `is_dome_restarting()`.

Finally, there is a computational loop. This loop must be called directly from `main()`, and the loop termination function, `loop_done()` here, must be defined entirely in terms of Dome variables. Thus, if this loop is reached and the values of all Dome variables have been restored from the checkpoint file, the computation will resume exactly at the point that the last checkpoint was taken. At the end of the computational loop there is a call to `dome_checkpoint()`, which will checkpoint all the Dome variables in the environment. The `dome_checkpoint()` method only results in a checkpoint file being saved every nth call where n is set by a command-line parameter.

This method effectively eliminates the need to restore the program counter. Since the state of the computational loop at the start of an iteration is defined entirely in terms of Dome variables and the checkpoint occurs at the end of an iteration, the computational loop can be resumed from the top on restart. Also, since the computational loop

```
main(int argc, char *argv[])
{
   // All Dome programs start by initializing the Dome environment.
   // Based on the arguments passed to the initialization method,
   // Dome determines whether the program is to start from the
   // beginning or restart from a saved checkpoint.
   dome_init(argc, argv);

   // Declare variables - Without the checkpointing package, the
   // dScalar<int> and dScalar<float> variables would simply be
   // declared as int and float variables.  They are declared as
   // dScalars here so that they will be included in the checkpoint.
*  dScalar<int> integer_variable;
*  dScalar<float> float_variable;
   dVector<int> vector_of_ints;
   dVector<float> vector_of_floats;
   // etc.

   // initialization code - should be skipped if in restart mode.
*  if (!is_dome_restarting())
        execute_user's_initialization_code(...);

   // computational loop
   while (!loop_done(...)) {  // loop_done is a function of dome vars
      do_computation(...);
*     dome_checkpoint();
      }
   }
```

Figure 1: Skeleton program following the model required for application level checkpoint/restart. Lines marked with an asterisk are those that have been added or changed to allow operation of the fault tolerance package in the given Dome program

is called from `main()`, no stack information needs to be saved. The variables are simply restored from the checkpoint file as they are declared, and on entry to the computational loop normal program execution can resume.

Of course, this is a limited programming model. In particular, the requirement that the computational loop be called directly from `main()` may seem extremely restrictive. However, it has been observed in [8] that a majority of scientific programs are either fully or loosely synchronous, that is, all processes repeatedly execute a section of code and then synchronize. Therefore, a large proportion of these applications can easily be adapted to this model. Only minor changes were required, for example, to fit the Dome molecular dynamics application to this model. The benefits of checkpoint and restart in a large application easily outweigh the small one-time cost of performing this adaptation.

2.3 Preprocessor-Based Application Level Fault Tolerance

The previous section described a restricted programming model for application level fault tolerance. The restrictiveness of that model simplifies the difficult task of saving and restoring the stack and the program counter. This section describes the use of a preprocessor to allow checkpoints to be taken at almost any point in the Dome application.

To illustrate the problems solved by preprocessing, examine the program fragment in Figure 2. In order to restore the program to resume execution from the checkpoint in `g()`, two things must be done. First, the program counter must be set to the point just beyond that checkpoint to execute `do_g_stuff_2`. Second, the stack must be in such a state that when g() exits, control returns to the point in f() immediately following the call to g() so that **next_statement** is executed. This is difficult to accomplish by providing simple library functions and presents the programmer with the daunting task of properly inserting the mechanisms to save and restore program counter and stack information. By using the preprocessing technique, many of the restrictions on programming style described in the previous section can be removed. It is still the programmer's responsibility to place the calls to **dome_checkpoint()** in the program, however the insertion of a checkpoint call is much simpler than the process required for using the library-based checkpointing technique. The programmer must also determine which program variables must be declared as Dome variables so that their values will be saved at the time of a checkpoint.

```
f() {
  dScalar<int> i;
  do_f_stuff;
  g(i);
  next_statement;
  ...
}

g(dScalar<int> &i) {
  do_g_stuff_1;
  dome_checkpoint();
  do_g_stuff_2;
}
```

Figure 2: A program fragment before checkpoint preprocessing.

The checkpoint preprocessor, however, can modify a Dome program automatically so that sufficient information is saved to restore the stack and program counter along with the required program variables upon restart. To accomplish this the preprocessor inserts sufficient labels and goto statements into the code to enable the program to visit every variable declaration and function call quickly, without executing any of the application's other code, until the state has been fully restored. Figure 3 shows the code fragment from Figure 2 after preprocessing.

Before each procedure call that could lead to a checkpoint, such as the call to g() in f(), a call to **dome_push()** is inserted. This call pushes an entry onto the procedure call stack which is maintained within the Dome environment. The stack entry contains both the name of the procedure being called and a unique sequence number. The sequence number is necessary if there are multiple calls to the same procedure on the stack. On restart conditional goto statements ensure that the program state is restored to the position of the last checkpoint. Until the position of the last checkpoint is reached, the only statements executed are variable declarations to restore the Dome variables, procedure calls to restore the stack, and goto statements to restore the program counter. The percentage of procedure calls that can lead to checkpoints is very likely small. Therefore, code inserted by the preprocessor will not significantly increase the size or complexity of the program.

Clearly the preprocessing method provides a much more flexible programming model than that described in the previous section. The application programmer is still responsible for inserting one or more calls to **dome_checkpoint()** in the program. However, this can be done almost anywhere in the code and is no longer restricted to a simple computational loop called from main().

An implementation of the preprocessor described here has been completed and has been successfully tested on Dome programs. Currently, more complex applications using Dome are being developed. These applications will be used for further testing of the checkpointing methods described here.

An alternate method of checkpoint preprocessing could be implemented in which a call to **dome_checkpoint()** would be inserted by the preprocessor at the beginning of every procedure. This method would be completely user-transparent but would be likely to create considerable additional overhead.

```
  f() {
    dScalar<int> i;

*   if (is_dome_restarting()) {
*        next_call = dome_get_next_call();
*        if (next_call == ``g1'') goto g1;
*        ...
*    }

    do_f_stuff;

*   dome_push(``g1'');
* g1:
    g(i);
*   dome_pop();

    next_statement;
    ...
  }

  g(dScalar<int> &i) {
*   if (is_dome_restarting())
*       goto restart_done;

    do_g_stuff_1;
    dome_checkpoint();

* restart_done:
    do_g_stuff_2;
  }
```

Figure 3: Program fragment after checkpoint preprocessing. Lines added by the preprocessor are marked with an asterisk.

2.4 Implementation Details

As can be seen from the previous examples, Dome programs are written in an SPMD style. A single C++ source program is replicated and executes in each parallel task. Dome is implemented as a library of C++ classes, and the underlying process control and message passing is provided by PVM. This makes the system very portable. To date it has been ported to eight architectures, including one MPP, the Intel Paragon.

Besides using a library of C++ objects, Dome programs must conform to a particular programming style. For instance, Dome programs should not utilize the underlying message passing system explicitly. Dome objects must be the only source of messages if the checkpointing is to function properly. Another minor restriction is that the checkpoint method must be invoked in all tasks. Thus, the following code is invalid because it is likely that the checkpoint method would only be called by a subset of the Dome tasks.

```
if (rand() % 2) dome_checkpoint();
```

The initial Dome task will spawn several copies of itself on nodes across the network of machines to be used. The set of machines is defined by the underlying PVM system, i.e., the *virtual machine*. Dome objects typically consist of some large amount of data that is fragmented among the Dome tasks. When a Dome object is declared, the constructor for the object is invoked in all the Dome tasks currently executing. Based on parameters to that constructor, the data in the object will be mapped to the Dome tasks. In most cases this requires no communication.

As the program runs, Dome records its execution rate on each machine. At predefined intervals, this information is exchanged and load balancing may be invoked. In the load balancing phase, Dome objects are re-mapped to match the past performance of the various machines. Some Dome operations require communication (such as a global sum) while others do not (pairwise binary operators, for example). Whenever a Dome object is created it registers with the Dome environment. This allows the Dome system to keep track of the distribution of data across the virtual machine. Furthermore, checkpoint and restore methods are defined for all Dome objects. The checkpoint method for each Dome object is called when the `dome_checkpoint()` method is invoked. At this point the Dome objects write out their internal state in XDR [19] format. In the preprocessor case the stack is also written to the checkpoint file. Upon restart the Dome object constructors test a global restart flag, and initialize their state from a checkpoint file if the flag is set. This allows the program state to be restored regardless of the architecture upon which the restart is occurring. Since the constructors dynamically perform the initial data mapping, the number of nodes at restart can differ from the number of nodes in use when the checkpoint was taken.

In the current implementation a master task is used to write a single checkpoint for the distributed program. At the checkpoint call, all the nodes write to a Dome I/O stream which is routed to the master task. The master task then writes the checkpoint file to a shared file system. Each checkpoint file created has a unique name based on the number of checkpoints previously performed for this program. This naming convention prevents a checkpoint which fails before completing from overwriting a previously created valid checkpoint. A special end-of-checkpoint marker is placed at the end of a completed checkpoint file. Upon restart, the most recent file with its end-of-checkpoint marker intact is used for restart.

2.5 Failure Daemon

In order to provide complete checkpoint/restart functionality, a failure daemon has been created for use with Dome. This daemon, known as the cleaner, monitors active Dome programs. When a failure of one of the tasks of a Dome program is detected, the cleaner attempts to restart the Dome program from the last valid checkpoint.

Only one cleaner process is run per PVM virtual machine, and that cleaner can monitor any number of concurrent Dome programs. When a Dome program begins it checks PVM's pvmd database for for an entry indicating the task id of the cleaner. If there is no entry or the task id given is no longer valid, that process will spawn a cleaner. The Dome program then registers with the cleaner, sending to the cleaner its program name, all command line arguments with which it was started, and the task ids that constitute that Dome program.

The cleaner, when started, first determines whether another cleaner is running. If there is already a cleaner process running due to a race condition in the startup, the new cleaner exits. Otherwise, the new cleaner registers with PVM, sets its task id in the pvmd database for other Dome programs to access, and waits for a message from a Dome program. The first message that the cleaner should receive is a Dome program registration message. When this message is received, the cleaner stores the program name, command line arguments, and task ids sent by the Dome program. It then issues a `pvm_notify()` for all tasks in that Dome program so that it will receive a Dome task failure message if any of those tasks terminates.

If the cleaner receives a Dome task failure message from the PVM daemon it issues a series of `pvm_kill()` commands to terminate all other tasks that are part of the same Dome program. It then finds the most recent valid checkpoint file for this program, and builds a command line from the Dome program name and arguments that were received when that program registered with the cleaner. The argument `-dome_restart` is added to (or modified in) the command line to indicate the checkpoint file from which to restart the Dome program. The cleaner then resubmits the Dome program with that command line to restart from the given checkpoint file.

When a Dome task completes successfully, it sends a Dome task complete message to the cleaner. Upon receipt of this message the cleaner removes this task from the list of tasks being monitored.

The cleaner process has been implemented to restart the whole Dome program rather than just the individual failed process. This procedure avoids the complications of rolling back existing processes.

3 Qualitative Comparison

A number of metrics can be used to compare the various levels of checkpointing. These metrics include usability, checkpointing cost, and portability.

System level checkpointing is clearly the easiest method to use. It is completely transparent to the application programmer who does not have to make any program changes. Application level checkpointing with preprocessing,

Level	Transparency	Portability	Costs
Application	very weak transparency	very portable code and checkpoints	produces small checkpoints
Application with Preprocessor	almost transparent (but may interfere with debugging)	very portable code and checkpoints	produces small checkpoints
System	completely transparent	neither code nor checkpoints portable	produces large checkpoints

Table 1: Comparison of levels of checkpointing.

however, is only marginally more difficult to use. The user faces some restrictions, and debugging can be slightly more difficult, but it is still relatively transparent. Without preprocessing considerable effort may be required of the programmer in adapting an application to the model. A large proportion of scientific applications, however, are expected to fit the model well.

The cost of a fault tolerance package can be measured in a number of ways. The most important cost is probably the added execution time of checkpoint creation. This will be examined empirically in Section 4. An important related metric, however, is the size of the checkpoint files. The creation of large files can significantly increase the cost of checkpointing. The system level checkpointing method produces very large checkpoints since, at best, it provides page level data granularity. System level methods also tend to save a large quantity of other information such as the complete stack contents. With the application level methods, however, only the Dome variables and a small amount of stack information are saved. This will produce significantly smaller checkpoint files. The Dome molecular dynamics application, for example, running as a single process creates a 10KB file with the application level checkpoint method. Using code from [14], the system level method, however, yields a 3.3MB file.

Finally, the application level checkpointing methods have a distinct advantage over the system level methods since they use ordinary C++ code and do not require any system-specific details. Thus, they are portable between architectures. In addition, the checkpoints created by the application level mechanisms can be used to restore program execution on any cluster of machines regardless of the architecture. Figure 1 summarizes the advantages and disadvantages of these checkpointing methods.

4 Checkpointing Results

This section discusses the results obtained thus far using application level checkpointing with Dome. It first presents a set of predicted results for the average total runtime of applications using this checkpointing method. These predictions use a formula based on the time to checkpoint, the total runtime if there are no failures, and an estimate of the failure rate. Then a set of actual measured runtimes are presented for checkpoint and restart of the Dome molecular dynamics application using the application level mechanisms described. These timings were collected for various failure rates and checkpointing frequencies.

4.1 Predicted Results

Most of the research in program checkpointing has focused on measuring the overhead of creating a checkpoint. A fault tolerance package is used, however, when failures are anticipated. Checkpointing in this situation ultimately reduces the time required for a program to run to completion. Thus, it is more useful to calculate the average total runtime of the application which is based on the checkpoint overhead and the failure rate. As discussed in [23], a Poisson distribution serves as an accurate model for the expected failure rate during a computation if the following assumptions are made: first, the chance of a failure occurring in any sufficiently small time interval is proportional to the size of the interval, second, the probability of multiple failures in the cluster in a given time interval is much smaller than the chance of a single failure, and, third, the probabilities of failures in non-overlapping intervals are independent. These seem to be a reasonable set of assumptions for making estimates of machine failure rates.

Duda [5] has used this formulation to calculate the expected runtime of a program as follows:

$$t = \frac{T}{a}(C + (C + \frac{1}{\gamma})(e^{(\gamma a)} - 1)) \tag{1}$$

where

$t =$	total expected time for a run
$T =$	program runtime in the absence of system failures
$a =$	uninterrupted program execution time between checkpoints
$C =$	time to create a checkpoint
$\gamma =$	Poisson parameter, or 1/(mean time between failures)

The time required to detect the presence of a failure and recover from it is assumed to be negligible for now. Using this formula the expected runtime of a program can be computed based on the time to checkpoint, the total runtime if there are no failures, and an estimate of the failure rate. For additional mathematical treatments of program runtimes in the presence of failures, see [11, 26].

In order to get a general idea of the costs of our checkpointing package, short, successful runs of the Dome molecular dynamics application, *md*, were timed. The *md* application is based on a CM-Fortran program developed at the Pittsburgh Supercomputing Center. It simulates the interactions of a bounded set of argon atoms over a set of discrete time intervals. For this experiment the application was distributed on eight Dec Alpha workstations and timed while checkpointing at various frequencies. Then several possible values for the mean time between failures were used to compute the total expected time for a run of *md* in the presence of system failures using Eq. 1. Distributing *md* across eight machines, simulating 500 atoms for 700 iterations, checkpointing incurred a cost of only about 0.2% per checkpoint in a 26 minute run. Figure 4 plots the calculated runtime for various values of the mean time between failures based on Eq. 1 and actual program runtimes measured in the failure-free case. It is interesting to note that if a failure occurs every three minutes and sufficient checkpoints are taken, the expected runtime will not even double. Furthermore, if the same number of checkpoints are taken in a failure-free run, there is only an additional 3% cost in the total runtime.

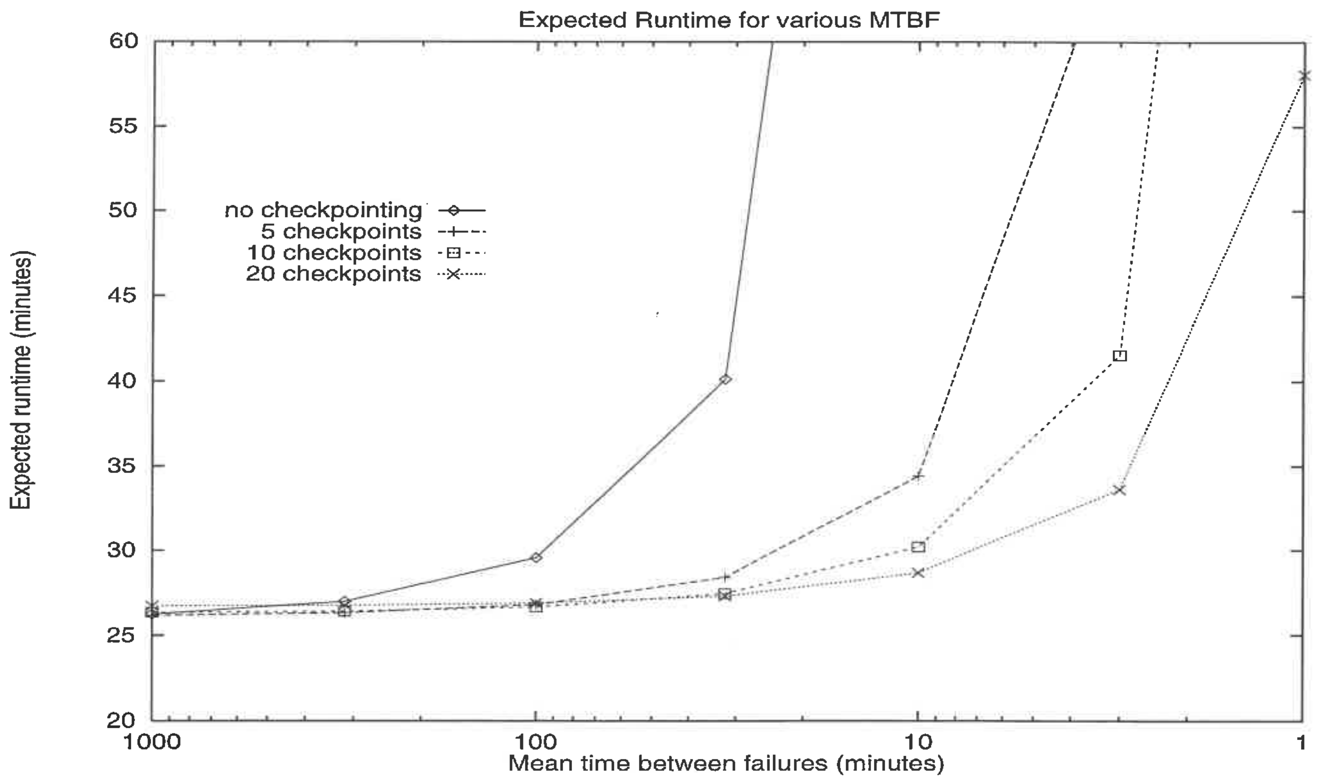

Figure 4: Expected runtime for *md* vs. mean time between failures.

In short program runs like the experiment described here, a system failure would not be expected. The results, however, scale well since the cost per checkpoint is constant for a given problem size. In *md* the computation complexity grows faster than the data size, so the checkpoint cost compared to the total runtime decreases. Thus, the runtimes for the checkpointing options given in Figure 4 should remain proportional for longer runs. In fact, the checkpointing cases would show additional improvement since the time to checkpoint is proportional to the problem size not the runtime.

In an experiment measuring the failure rates of systems on the Internet, Long, Carroll, and Park [18] found that, depending on the system, the mean time between failures tended to be between 12 and 20 days. If 16 days is used as a rough estimate, a cluster of eight machines is likely to have a failure every 2 days on average. Since the results plotted in Figure 4 should remain proportional for longer runs, the units of the graph could reasonably be changed from minutes to days. Then, given a mean time between failures of 2 days, a properly chosen checkpoint interval would increase the expected runtime by only about 12 days over the failure-free case of about 25 days. Without a checkpoint and restart mechanism such a program would probably never complete successfully.

4.2 Empirical Results

In order to evaluate the effectiveness of the checkpointing methods described here, the *md* program was run with various checkpointing frequencies and failure rates. To simulate the various failure rates, the cleaner process was modified to induce failures in a distributed Dome program based on a Poisson distribution. This was done by calling `alarm()` with a calculated failure time and issuing a `pvm_kill()` for one of the tasks of the Dome program when the SIGALRM signal was received.

The experiments described here were run on eight dedicated DEC Alpha workstations connected by a DEC Gigaswitch. The load balancing option of the Dome system was not used. To simplify the experiment, after an induced failure the program was restarted on the same virtual machine that it had been using prior to the failure. This prevented the cleaner process from having to maintain a pool of available machines to add to the PVM virtual machine after each failure. It is interesting to note, however, that, although measurements of this case were not made, the architecture of Dome would have allowed the program to be restarted on fewer machines than it had previously been running on if that number of machines were no longer available after the failure.

Figure 5 plots the observed runtime for various mean times between failures and checkpointing frequencies. Unlike the predicted results presented in Section 4.1, the completion times plotted here were measured for actual program runs in the presence of failures. Failures were induced in the application by the cleaner process based on Poisson distributions using varying values for the mean time between failures. Given the assumptions discussed in Section 4.1, a Poisson distribution with a mean time between failures of L for sufficiently small intervals of size h gives a probability of failure of approximately h/L. One second was chosen in this experiment as the interval over which to test for the probability of a failure. Therefore, since the probability of a failures in disjoint intervals is independent, our cleaner process simply loops through the succeeding seconds in the run and, using the `random()` function, calculates the next failure time for that run with the given mean time between failures.

The sharp "L" shape of the **no checkpointing** curve in Figure 5 indicates that the program cannot recover after a failure because our checkpoint and restart mechanisms are not in operation for those runs. Comparing figures 4 and 5 one can see that the actual behavior of the system closely adheres to our model.

There are several factors which contribute to the overhead of this checkpoint/restart method. These include the time to create a checkpoint file, the time between the failure and recognition of that failure, the time required to kill the remaining tasks and resubmit the program, the time to restore and redistribute the data from the checkpoint file, and the time required to re-execute steps performed from the time of the last checkpoint to the time of failure. As was previously noted, the time to create a checkpoint file was measured at 0.2% of the total runtime in *md*. Checkpoint creation time, however, is a function of the number and size of Dome variables used in the application and, therefore, will vary from application to application. The failure recognition time was measured as the time from the cleaner inducing failure to the receipt of a task failure notification from PVM. This time averaged 52.4 milliseconds in these runs. The time to issue a `pvm_kill()` for each of the remaining processes and resubmit the job to restart from the appropriate checkpoint file averaged 103.9 milliseconds. Like the checkpoint file creation time, the time to restore and redistribute checkpointed data is a function of the application. In the measurements of *md* collected, this cost was seen to be approximately 2% of the total runtime in the case of a single restart.

Any checkpoint and restart package to be used with the Dome system must be highly portable. The portability of our checkpointing package has been demonstrated by running *md* with checkpointing on both DEC Alpha and SGI

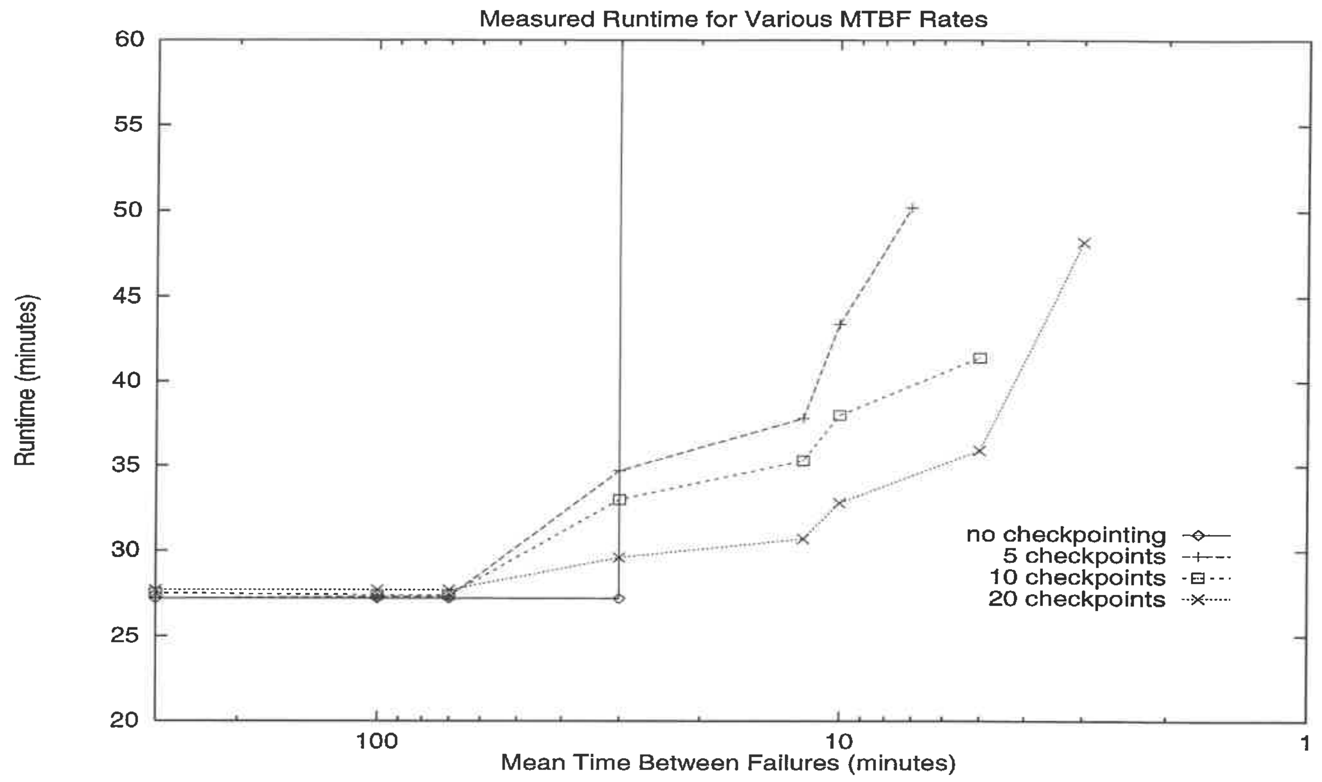

Figure 5: Measured runtime for *md* vs. time between failures.

workstations. Furthermore, the portability of the checkpoints themselves has also been successfully accomplished by restarting *md* on Alpha workstations from checkpoints created on SGI workstations.

5 Integration with Condor

We have created a prototype implementation of Dome that is integrated with the Condor [17] system. Condor provides mechanisms for the migration of sequential processes among a pool of monitored workstations. This allows for efficient use of idle cycles in a NOW.

Our integration of Dome with Condor leverages Condor's monitoring facilities to perform the initial distribution of the Dome tasks onto a number of workstations. The Dome program which is running uses the checkpoint and restart mechanism described in this paper to save periodic checkpoints. If the load characteristics of any of these workstations exceed a given set of parameters, Condor will terminate the Dome task that is running on that machine. The Dome cleaner process then recognizes that one of the Dome tasks has died, terminates the remaining Dome tasks, and resubmits the entire Dome program to Condor to be restarted from the most recent checkpoint file. Condor will restart the program on a set of machines which meet the specified load requirements.

This integration of Dome and Condor makes it possible to execute a Dome application on a number of workstations and then to migrate that Dome application to a different set of workstations when the load characteristics warrant. By using Dome's checkpoint mechanism the program can be migrated to a set of machines which differ *both in number and in architecture.* This malleability and flexibility can have a considerably positive effect on system throughput.

6 Related Work

A number of systems targeted at system level interrupt-driven checkpointing for parallel programs have been developed recently. Li, Naughton, and Plank [15, 16, 21] have concentrated on minimizing the overhead of the individual checkpoints by using system level techniques related to memory protection and designing special algorithms to take advantage of multicomputer architectures. Silva and Silva [24] have designed a system to account for the latency between failure occurrence and failure detection. Another system, designed by Leon, Fisher, and Steenkiste [14], is specifically tailored to checkpoint and restart multicomputer applications written in PVM. All of these systems depend on machine-specific code, however, and none provides checkpoints that are portable within a heterogeneous environment.

Silva, Veer, and Silva [25] have developed an application level checkpointing system for distributed programs. Their primary focus is also to minimize the cost of individual checkpoints. Some studies such as [6], however, have suggested that checkpointing generally tends to be an inexpensive operation. Therefore, our fault tolerance package focuses on other issues. The approach described in this paper uses object-oriented techniques to create a user-transparent and fully portable checkpoint and restart mechanism for distributed programs.

Huang and Kintala [13] have built a system that provides several levels of software fault tolerance for client/server applications. They demonstrate that these mechanisms are quite useful in providing appropriate system behavior in terms of availability and data consistency.

Plank et al. [22] have a unique approach which uses diskless checkpointing. They utilize a parity processor rather than a disk for storing processor state. Upon failure the parity processor is able to reconstruct the state of the failed processor from the parity and the state of the remaining processors. Application level information is used by the system to determine what state information to checkpoint. Their diskless checkpointing is integrated with the ScaLAPAK [7] linear algebra routines, allowing a user program to survive a fault that occurs while the program is within the instrumented ScaLAPAK library without incurring any disk access. If the program is to survive faults that do not occur while the program is executing the linear algebra library code, however, the programmer is required to save that state explicitly, perhaps using a traditional disk-based checkpointing mechanism. This work is similar to Dome's checkpointing mechanism in that it is integrated into a library and uses application level information.

Peercy and Banerjee [20] have a fault tolerance mechanism that is similar to Dome in that they use high level information about the application's structure. Their approach leverages the structure of the Actor model to allow them to keep shadow processors up to date with sufficient information that the processors can be quickly reconfigured. Their approach suffers when the application is communication bound because it doubles the number of data messages. In contrast Dome's mechanism takes a more conventional approach by saving the checkpoints to stable storage rather than relying on shadow processes.

Hofmeister and Purtilo [12] have written a preprocessor for saving the state of programs which use their Polylith system. While their primary focus is dynamic program reconfiguration rather than checkpoint and restart, their preprocessing method is somewhat similar to the one described here.

7 Future Work

There are a number of improvements to be made to our checkpointing system. Several of these improvements will help to reduce the checkpointing overhead further. These are described in studies such as [14, 15, 16, 24] and include checkpointing to memory rather than disk, using incremental or copy-on-write methods to reduce the amount of data saved, or creating "optimistic" local checkpoints at each process rather than synchronizing for a global checkpoint. The SPMD model makes this optimistic checkpointing very easy. Our failure daemon might also be expanded to handle the additional issues of I/O that are not currently addressed.

Investigation has also begun on the use of our preprocessing method for general PVM programs. Checkpointing is much more challenging in a general PVM application since the set of variables that need to be checkpointed and the current location in the parallel program may be different in each process. Thus, unlike the checkpoint of a Dome program, the checkpoint of a PVM application must consist of a set of files, one per process, and the application must be restarted with the same number of processes. Synchronization is required at checkpoint time to ensure a consistent state, and a "snapshot" algorithm such as the one described by Chandy and Lamport [4] must be used to ensure that no messages are in transit. While this might be more expensive, it retains the advantage of completely machine-independent checkpoints, which should be very useful to PVM programmers.

8 Conclusions

Application level checkpointing has proven to be a useful method for implementing fault tolerance in SPMD parallel programming systems like Dome. Application level fault tolerance sacrifices some user transparency but provides complete portability of both the checkpointing code and the checkpoints themselves. Experiments have suggested that while incurring very small checkpointing costs, significant savings can be realized in the total expected runtime of a computation in the presence of failures. As our work on reducing the overhead and programmer effort required while increasing the efficiency and applicability of our system continues, our fault tolerance features will be a significant benefit to Dome programmers and are expected to lead to similar mechanisms for general message passing programs.

9 Availability and Acknowledgements

The Dome system, including the checkpoint and restart mechanisms described in this paper, were released in May of 1996. More information can be found on the Dome home page, `http://www.cs.cmu.edu/~Dome`.

We would like to acknowledge Dennis Gannon's group for their work on the Sage++ compiler preprocessor toolkit [3]. This toolkit was used to build the preprocessor of Dome programs described in Section 2.3.

References

[1] José Nagib Cotrim Árabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Michael Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CMU-CS-95-137, Carnegie Mellon University, April 1995.

[2] José Nagib Cotrim Árabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Michael Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. In *International Parallel Processing Symposium 1996*, Honolulu, HI, April 1996.

[3] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *OONSKI'94*, 1994.

[4] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] Andrzej Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229, June 1983.

[6] Elmootazbella Elnozahy, David Johnson, and Willy Zwaeneopoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE Computer Society Press, 1992.

[7] J. Choi et al. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *4th Symposium Frontiers of Massive Parallel Computers*, pages 120–127, 1992.

[8] Geoffrey C. Fox. What have we learned from using real parallel machines to solve real problems? In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 897–955. Association for Computing Machinery, 1988.

[9] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[11] Erol Gelenbe. On the optimum checkpoint interval. *Journal of the Association for Computing Machinery*, 26(2):259–270, April 1979.

[12] Christine Hofmeister and James Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. Technical Report UMIACS-TR-92-120, University of Maryland, November 1992.

[13] Yennun Huang and Chandra Kintala. Software Implemented Fault Tolerance: Technologies and Experience. In *FTCS-23*, June 1993.

[14] Juan Leon, Allan Fisher, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[15] Kai Li, Jeffrey Naughton, and James Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88. Association for Computing Machinery, 1990.

[16] Kai Li, Jeffrey Naughton, and James Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 2–11. IEEE Computer Society Press, 1991.

[17] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[18] D.D.E. Long, J.L. Carroll, and C.J. Park. A study of the reliability of Internet sites. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 177–186. IEEE Computer Society Press, 1991.

[19] Sun Microsystems. XDR: External Data Representation Standard. RFC 1014, June 1987. 20 pages.

[20] Michael Peercy and Prithviraj Bannerjee. Software Schemes of Reconfiguration and Recovery in Distributed Memory Multicomputers using the Actor Model. In *FTCS-25*, pages 479–488, Pasadena, CA, June 1995.

[21] James Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technology*, pages 62–66, Summer 1994.

[22] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations. In *FTCS-25*, Pasadena, CA, June 1995.

[23] Sheldon Ross. *A First Course in Probability*. Macmillan Publishing Company, 1988.

[24] Luis Silva and João Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162. IEEE Computer Society Press, 1992.

[25] Luis Silva, Bart Veer, and João Silva. Checkpointing SPMD applications on transputer networks. In *Proceedings of the Scalable High Performance Computing Conference*, pages 694–701, May 1994.

[26] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.