High-Level Fault Tolerance in Distributed Programs

Erik Seligman Adam Beguelin

December 1994

CMU-CS-94-223

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We have been developing high-level checkpoint and restart methods for Dome (Distributed Object Migration Environment), a C++ library of data-parallel objects that are automatically distributed using PVM. There are several levels of programming abstraction at which fault tolerance mechanisms can be designed: high-level, where the checkpoint and restart are built into our C++ objects, but the program structure is severly constrained; high-level with preprocessing, where a preprocessor inserts extra C++ statements into the code to facilitate checkpoint and restart; and low-level, where periodically an interrupt causes a memory image to be written out. Because we consider portability (both of our libraries and of the checkpoints they produce) to be an important goal, we focus on the higher-level checkpointing methods. In addition, we describe an implementation of high-level checkpointing, demonstrate it on multiple architectures, and show that it is efficient enough to provide good expected run times with low overhead, even in the case of frequent failures.

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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, CNRI, ARPA, or the U.S. Government.

Keywords: Heterogeneous parallel programming, checkpoint and restart, fault tolerance, Dome, PVM

1 Introduction

Using large clusters of workstations, rather than huge and expensive supercomputers, to solve scientific problems is a current focus in the high performance computing field. While this method offers many advantages, it also creates some problems, including the issue of how to handle failures on some subset of the nodes. 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, if we have a workstation with a mean time between failures of 16 days, a one day computation will have a 94% chance of a successful run; but on a cluster of ten such machines, there is only a 54% ($.94^{10}$) chance that the computation will be complete before one of them fails. Thus it is vital that some kind of fault tolerance mechanism be incorporated into any system designed for extended execution on a workstation cluster. Here we discuss the fault tolerance mechanisms we have been building for Dome (Distributed Object Migration Environment [1]), an object-oriented library we have developed in which objects, implemented in C++, automatically distribute themselves using PVM [5]. We have been experimenting with fault tolerance at various levels of programming abstraction, and this paper describes the results we have achieved in our initial implementation.

A Dome program is a C++ program that instantiates objects defined in our Dome library. The programming model is SPMD (Single Program Multiple Data)— each machine in the cluster runs a separate copy of the same Dome program, and the methods defined for Dome objects use PVM to handle distribution of the subelements and communicate when required. From the programmer's point of view, a Dome application is simply a data-parallel program. For a full discussion of Dome, see [1].

There are several levels of programming abstraction at which one could implement a fault tolerance package in Dome. At the highest level, we have provided a set of C++ methods which can be called to checkpoint the program's data structures. Another option is to do the checkpointing with C++ code, but to have a preprocessor (effectively, a compiler front end) insert most of the checkpointing calls automatically, saving the user some work. Finally, we could use a truly transparent low-level checkpointing package, so the user has little work to do; the system saves a memory image periodically upon interrupt, from which it is easy to restore the state later. In this case, determining a consistent state (messages may be in transit) is a major issue, while in the higher-level methods the knowledge of the program structure makes this problem much simpler.

While high-level fault tolerance tends to require more work from the users, we believe it is an important feature to implement in Dome, since one of our major goals is for Dome to be easily portable to any system that supports PVM and C++. Since we want to run Dome on heterogeneous systems, furthermore, it is vital that checkpoints created on one architecture be usable on others. What if the only Cray was the machine that failed? Thus we have concentrated on developing usable high-level checkpointing features for Dome.

We have completed an implementation of high-level checkpointing and timed it running on *md*, a molecular dynamics application we have written, based on a CM-FORTRAN program developed at the Pittsburgh Supercomputing Center, using Dome. Our timings show that even in the case of frequent failures, our checkpointing overhead is low enough to provide a good expected run time for this application.

2 Related Work

A number of systems targeted at low-level interrupt-driven checkpointing have been developed recently. Li, Naughton, and Plank ([9], [10], [12]) have concentrated on trying to minimize the overhead of the individual checkpoints by using system-level techniques related to memory protection, and designed special algorithms to take advantage of multicomputer architectures. Silva and Silva [14] have designed a system to take into account the latency between failure occurrence and failure detection. Another system being designed by Leon, Fisher, and Steenkiste [8] is specifically tailored to checkpoint and restart multicomputer applications written in PVM. All of these systems depend on machine-specific code, and none of them provide checkpoints that are portable between heterogeneous systems.

Silva,Veer,and Silva [15] have also developed a high-level checkpointing system for distributed programs. Their primary focus was in attempting to minimize the cost of individual checkpoints. Some studies, such as [3], however, have suggested that in general, checkpointing tends to be a cheap operation, and we want to take advantage of this fact to concentrate on other issues. We are mainly concerned with using the object-oriented paradigm to maximize the user-transparency of a fully portable checkpoint and restart mechanism for distributed programs.

Hofmeister and Purtilo [7] have written a preprocessor for saving the state of programs written using their Polylith system. While their main concern is dynamic program reconfiguration rather than checkpoint and restart, their preprocessing method is somewhat similar to the one we describe below.

3 The Model

It is important to define exactly what we mean by a fault tolerance package. For our purposes, we are assuming that all failures will be catastrophic; i.e., either the program runs to completion and produces the correct results, or the run terminates prematurely and informs the user of this fact. We are not allowing for the possibility of arbitrary, or Byzantine, failure modes which may corrupt the results without the user's knowledge. Thus, we are attempting to provide a package that periodically saves the program's current state to one or more checkpoint files, and allows the program to be restarted from the most recent checkpoint after a failure. The restarted program should then proceed as if the failure had never occurred.

It should be noted that there are a number of issues related to checkpointing which we are not dealing with currently. We are assuming that a checkpoint is a global operation— all processes synchronize and a central process writes the checkpoint to a file, which is guaranteed not to be lost or destroyed by a failure. In addition, we are restarting the whole program rather than individual processes, to avoid the complications of rolling back existing processes. We are also not dealing with issues of how to checkpoint the outside world— if the program reads and writes to a file, for example, or requires periodic input from a human user, it must be ensured that we can somehow "rewind the world" to its state at the last checkpoint; but we are assuming for now that we will be using applications where this is not an issue. Note that these are not fundamental limitations of our approach, but issues that we are planning to address in future work.

Most research on checkpointing tends to focus on measuring the overhead of taking the checkpoints, but we do not believe this is the proper measure to take. The reason we do checkpointing is that when failures are anticipated, checkpointing reduces the time we expect to wait for a program to run to completion. Thus we want to calculate the average total run time, based on the checkpoint overhead and the failure rate. In order to estimate this expected run time, we will make some simplifying assumptions. If the chance of some failure occurring in any sufficiently small time interval is proportional to the size of the interval, the probability of more than one failure on a single time interval is much smaller than the chance of one failure, and the probabilities of failures in nonoverlapping intervals are independent, then, as discussed in [13], a Poisson distribution will be an accurate model of the expected number of failures during a computation.

Using this formulation, Duda [2] has calculated the expected run time (we ignore the time to detect and recover from a failure, which we are assuming to be negligible for now):

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

where

$t =$	total expected time for a run
$\gamma =$	Poisson parameter, or 1/(mean time between failures)
$T =$	run time if there are no failures
$a =$	T/(number of checkpoints)
$C =$	time to create a checkpoint

Using this formula, we can figure out the expected run time of a program based on the time to checkpoint, the total run time if there are no failures, and an estimate of the failure rate. For additional mathematical treatments of program run times in the prescence of failures, see [6], [16].

4 The State of the System

Before discussing the details of our implementation, it is also necessary to detail what we mean by the "state" of a computation. In general terms, this consists of the user memory, stack, registers, messages, and the relevant operating system variables. At checkpoint time, though, we want to save just enough information to successfully resume the program later.

Thus, we need to figure out exactly what information we would need in order to restart a program from a saved checkpoint. Remember that by "high level" we mean that our checkpoint/restart mechanism must operate entirely through the use of C++ code, and not peek into any internal machine information such as register and stack values. The relevant state (for our purposes) consists of:

- **The program counter.** We need to be able to figure out exactly where in the program we were when the checkpoint occurred. This is hard to do in a high-level checkpointing mechanism; we will simplify the problem by guaranteeing that a checkpoint will only occur at a user-inserted call to a `dome::checkpoint()` method. Even then, we must restrict the programming model so that it is somehow possible for the control flow to arrive quickly at this checkpoint call upon restart.

- **The stack.** There will be some set of procedure calls active at the checkpoint, and we need to rebuild the stack so we 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 actual values on the internal stack, and even if we can save it, reconstructing it upon restart is not easy.

- **Some subset of the variables**. Note that we do not say "all the variables in the program", since in most programs, many are temporary and do not need their values preserved except during certain parts of the execution. Remember that since we are talking about high-level checkpointing, program data can be viewed in terms of variables, rather than details like registers and memory contents.

  A problem we could have here is that the program may make use of pointers and user-allocated memory. For now, we are disallowing this. This is not as much of a limitation as it might seem, since C++ has a pass-by-reference mechanism which replaces many of the uses of pointers in C.

- **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 inside calls to methods in our Dome library; the user does not make explicit communication calls. Thus, as long as we require that the high-level checkpoints be taken between calls to Dome methods rather than inside them, we can guarantee that no communication state will need to be saved.

The next two sections, on the implementation of high-level checkpointing without and with a preprocessor, will discuss in more detail how we are dealing with the problems of checkpointing each of these components of the program state.

5 High Level Fault Tolerance

In high level checkpoint and restart, we want to operate entirely in C++ code without using machine-specific details, looking at the internal machine state, or requiring any special compiler or preprocessor. In order to do this sucessfully while minimizing the amount of code the user has to insert, we have imposed some severe restrictions on the available programming model. We have demonstrated that *md* can be easily adapted to fit this model, as, we believe, can a wide variety of scientific applications.

The model that a program must follow in order to make use of our high level checkpointing package is illustrated by the skeleton program in figure 1.

The model works as follows. First, the Dome environment must be declared; based on the arguments to the program, a flag may be turned on that indicates the environment is restarting. Next the Dome variables, consisting of `dVectors` and `dScalars`, must be declared. Note that `dVectors` are automatically distributed by the Dome library, while `dScalars` are ordinary variables, duplicated at every process. For all practical purposes, a `dScalar<int>` is the same as an `int`, a `dScalar<float>` as a `float`, etc. The reason for using these is that they are registered with the Dome environment, which has pointers to them for use during checkpoint and restart. Thus, if the user wants to checkpoint and restart a variable of type `T`, they should declare it as a `dScalar<T>`. (We have found the overhead of using `dScalars` to be only about 1-2% in *md*.) If the restart flag is on, then declaring a variable will actually cause its value to be read in from the restart file.

Once the variables have been declared, they either are ready to be initialized (if we are not restarting from a checkpoint file) or have had their values restored and are ready to be used. Therefore we must next have an initialization section, which we skip if Dome is in restart mode.

Finally, we have a main loop. Note that we require this loop to be in `main()`, and to be set up so that its state at the beginning of an iteration is entirely defined by the Dome variables. Thus if we reach this loop and have restored

```
main(argc,argv)
{
   // All Dome programs start by declaring a Dome environment.
   // Dome constructor decides if we are starting the program from
   // scratch or restarting from a saved checkpoint, based on argv.
   dome dome_env(argc,argv);

   // declare variables
   dScalar<int> integer_variable;
   dScalar<float> float_variable;
   dVector<int> vector_of_ints;
   dVector<float> vector_of_floats;
   // etc.

   // initialization code-- user must skip if in restart mode.
   if (!dome_env.is_restarting())
        execute_user's_initialization_code(...);

   // main loop
   while (!loop_done(...)) {  // loop_done is a function of dome vars
        do_computation(...);
        dome_env.checkpoint();
        }

}
```

Figure 1: Skeleton program following the model required for high-level checkpoint/restart.

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

g(dScalar<int>& i) {
  do_g_stuff_1;
  dome_env.checkpoint();
  do_g_stuff_2;
}
```

Figure 2: A program fragment, before preprocessing.

all the Dome variables to their values from the checkpoint file, the computation will proceed exactly as if it had been running from the saved checkpoint. At the end of the main loop there should be a call to `dome::checkpoint()`, which will checkpoint all the Dome variables in the environment. (Actually, only every nth call to this function results in a checkpoint, where n is determined by a command-line parameter.)

This method effectively eliminates the need for figuring out how to restore the program counter; since the state of the main loop at the start of an iteration is required to be entirely defined by its Dome variables, and the checkpoint is required to occur at the end of an iteration, we can just enter the main loop at the top once all the variables from the checkpoint file have been restored. Also, since we require the main loop to be in `main()`, there is no important information on the stack. The variables are simply restored from the checkpoint file as they are declared, and on entry to the main loop the program can continue as before.

Of course, this is a very limited programming model; in particular, the requirement that the main loop be at the top level might seem extremely restrictive. However, it has been observed [4] 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), so we believe that a large proportion of them could be adapted to follow this model without too much work. With only minor tweaking, for example, we have been able to translate *md* into this form. The benefits of being able to checkpoint and restart a large application will easily outweigh the one-time cost of performing this adaptation.

6 Preprocessing Method

In the previous section, we discussed a restricted programming model that is required for high-level fault tolerance. The main reason why this model has to be so restrictive is the difficulty of restoring the stack and the program counter. In order to illustrate this problem, let us examine the program fragment in figure 2.

Suppose we are trying to restore the program so it could continue from the checkpoint in `g()`. We need to not only get the program counter to the point just beyond that checkpoint (so we can execute `do_g_stuff_2`), but the stack has to be in such a state that when `g()` exits, we will return to the point in `f()` immediately following the call to `g()` and execute `next_statement`. This task is difficult to accomplish by simply providing library functions. This is our motivation for considering a preprocessor.

With a preprocessor, we could modify a Dome program automatically so that it will save enough information for the stack and program counter to be restored, along with the required program variables, upon restart. In order to do this, the preprocessor would insert sufficient labels and goto's 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.

For an example of how this would work, see figure 3. It is the same fragment as in figure 2, but after preprocessing.

Before each procedure call that could lead to a checkpoint (probably a small percentage of the total number of

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

*   if (dome_env.is_restarting()) {
*         next_call=dome_env.get_next_call();
*         if (next_call == ``g1'') goto g1;
*         ...
*    }

    do_f_stuff;

*   dome_env.push(``g1'');
* g1:
    g(i);
*   dome_env.pop();

    next_statement;
    ...
  }

  g(dScalar<int>& i) {
*   if (dome_env.is_restarting())
*       goto restart_done;

    do_g_stuff_1;
    dome_env.checkpoint();

* restart_done:
    do_g_stuff_2;
  }
```

Figure 3: Program fragment after preprocessing. Lines added by the preprocessor are marked with a '*'.

procedure calls), such as the call to `g()` in `f()`, a call to `dome::push()` is inserted, which pushes the procedure call onto a stack stored in the Dome environment, and the procedure call is labeled. Note that the label contains both the name of the procedure and a unique sequence number, in case there are multiple calls to the same procedure. Then on restart, the conditional goto's will make sure that until the program state is restored to what it was at the last checkpoint. The only statements executed are variable declarations (so the Dome variables can be restored), procedure calls (to restore the stack), and goto's (to restore the program counter). An alternative method of preprocessing which would be completely user-transparent would insert a checkpoint call at the beginning of every procedure; however, we believe that this would be likely to create unacceptable amounts of additional overhead, and it assumes that the inner loop contains at least one (non-library) procedure call.

Clearly the preprocessing method provides a much more flexible programming model than that described in the previous section. The user is responsible for inserting one or more calls to `dome::checkpoint()` in the program, but is free to do this almost anywhere, and is no longer restricted to a simple main loop at the top level. There are still some other limitations, such as the inability to use pointers to functions or user-allocated memory; we have not fully characterized these.

We have not yet implemented this method.

7 Qualitative Comparison

There are a number of metrics we can use to compare the various levels of checkpointing. There is the issue of usability; if a checkpointing method requires a lot of work from the user, it will never be used. Another important issue is the costs; aside from time, we also need to worry about the size of the checkpoints. Finally, we are concerned with the issue of portability: how hard is it to adapt each method to be used with a new architecture?

From the point of view of usability, it is clear that low-level transparent checkpointing is easiest to use, since the user does not have to do anything to the program. High level checkpointing with preprocessing, however, is only marginally more difficult to use; the user faces some restrictions, and debugging is made more difficult, but it is still relatively transparent. Without preprocessing, of course, considerable effort may be required of the users; but as we have mentioned, we believe a large proportion of scientific programs could be adapted without too much effort.

Looking at costs, the most important cost is probably the additional execution time, which we need to examine empirically. An important related metric, however, is the size of the checkpoint files; having to write large files could significantly increase the costs incurred by checkpointing. The low-level checkpointing method will produce very large checkpoints, since it provides at best page-level data granularity. Low-level methods also tend to save a large quantity of other information, such as the full stack contents. With the high level methods, however, we are only writing the Dome variables and a small amount of stack information, which in principle may produce significantly smaller checkpoint files. For example, using our high-level checkpoint method for *md* running on a single process yields a 10KB file, while a low-level method, using the code from [8], gives us a 3.3MB file.

Finally, in terms of portability, the high level checkpointing methods have a definite advantage over the low-level ones. The system itself is portable, since the high-level methods use ordinary C++ code and do not require any system-specific details, unlike the low level methods. In addition, the checkpoints saved can be used again in any Dome system, regardless of the architecture run on. This can be important when there is not a machine available of the same architecture as the one that failed, or when we desire to continue a saved computation on a different set of machines.

The chart in figure 4 summarizes the discussion in this section.

8 Empirical Results

In order to get a general idea of the costs of our checkpointing code, we timed short (successful) runs of *md* on eight Dec Alpha workstations, while checkpointing at various frequencies. We then used several possible values of the mean time between failures, and computed the total expected time for a run of *md* where failures are expected.

We found that when running *md* on eight machines, simulating 500 atoms for 700 iterations, checkpointing incurred a cost of about 0.2% per checkpoint, in a run that took about 26 minutes. Figure 5 graphs the expected runtime we calculated for various values of the mean time between failures. It is interesting to note that if we put in so many checkpoints that a failure every 2 minutes will not even double our expected runtime (rather than

Level	Transparency	Portability	Costs
High	very weak	very portable code & checkpoints	time? small size
High w/ Preprocess	almost transparent (but interferes with debugging)	very portable code & checkpoints	time? small size
Low	completely transparent	code & checkpoints not portable	time? larger checkpoints

Figure 4: Comparison of levels of checkpointing.

incurring the thousands-of-times expected cost without checkpointing), we only suffer a 3% cost to our runtime in the failure-free case.

Of course, the next question one should ask is what are realistic values for the mean time between failures. In a small run as in our experiment, we do not expect a failure at all, but it should be noted that our results would scale up nicely, since we found the cost per checkpoint to be a constant for a given problem size. Actually, for *md*, the computation complexity grows faster than the data size, so the checkpoint cost compared to the total run time will go down. Thus we would expect the graph in Figure 4 to remain valid for longer runs (that is, the units could be hours or days rather than minutes); in fact, the figure would overestimate the cost of checkpointing in runs with larger data sets.

In an experiment measuring the failure rates of systems on the Internet, Long, Carroll, and Park [11] found that depending on the system, the mean time between failures tended to be between 12 and 20 days. If we take 16 days as a rough estimate, a cluster of eight machines is likely to have a failure an average of every 2 days. Looking at the graph as if the units were days rather than minutes (realistic for some large simulations), we can see that a properly chosen checkpoint interval can result in less than a 50% increase in expected runtime over its ideal value, while without checkpointing such a program would take several millenia to run to completion.

Another experiment we have done is to demonstrate the portability of our checkpointing code by running *md* with checkpointing on both DEC Alpha and on SGI workstations. We have also successfully demonstrated the portablilty of the checkpoints themselves by restarting *md* on the Alphas from checkpoints created on the SGI's. We plan to demonstrate the portability of our system on additional architectures as our experiments continue.

9 Future Work and Conclusions

We are planning some more detailed experiments to get a much more precise picture of the actual benefits and costs of checkpointing. We would ideally like to run some very large simulations and observe the expected runtimes instead of calculating them, since studies such as [11] have suggested that failures may not in fact be a Poisson process. We also might be able to accomplish this by using actual observed failure times to create an accurate simulation. Furthermore, we are engaged in an ongoing effort to increase the number of Dome applications, in order to explore the results of high-level checkpointing in a wide variety of situations. We are also working on acquiring a Dome-compatible low-level checkpointing system for performance comparisons.

There are a number of improvements that we plan to make in our checkpointing system. First, we are going to complete the implementation of the high-level preprocessing method, which should combine the advantages we have found for high-level checkpointing with much better user transparency. There are also several improvements possible which would further reduce the checkpointing overhead, as described in studies such as [9],[10],[14],[8]. These include checkpointing to memory rather than disk, using incremental or copy-on-write methods to reduce the amount of data saved, or optimistically doing local checkpoints at each process rather than synchronizing for each checkpoint. (In fact, the SPMD model makes optimisitic checkpointing very easy, since each point in the computation has a globally known iteration number even without communication.) It would also be useful to implement a "failure daemon" method as in [8] to automatically detect failures and initiate recovery, and to expand our checkpointing system to handle additional issues like file I/O.

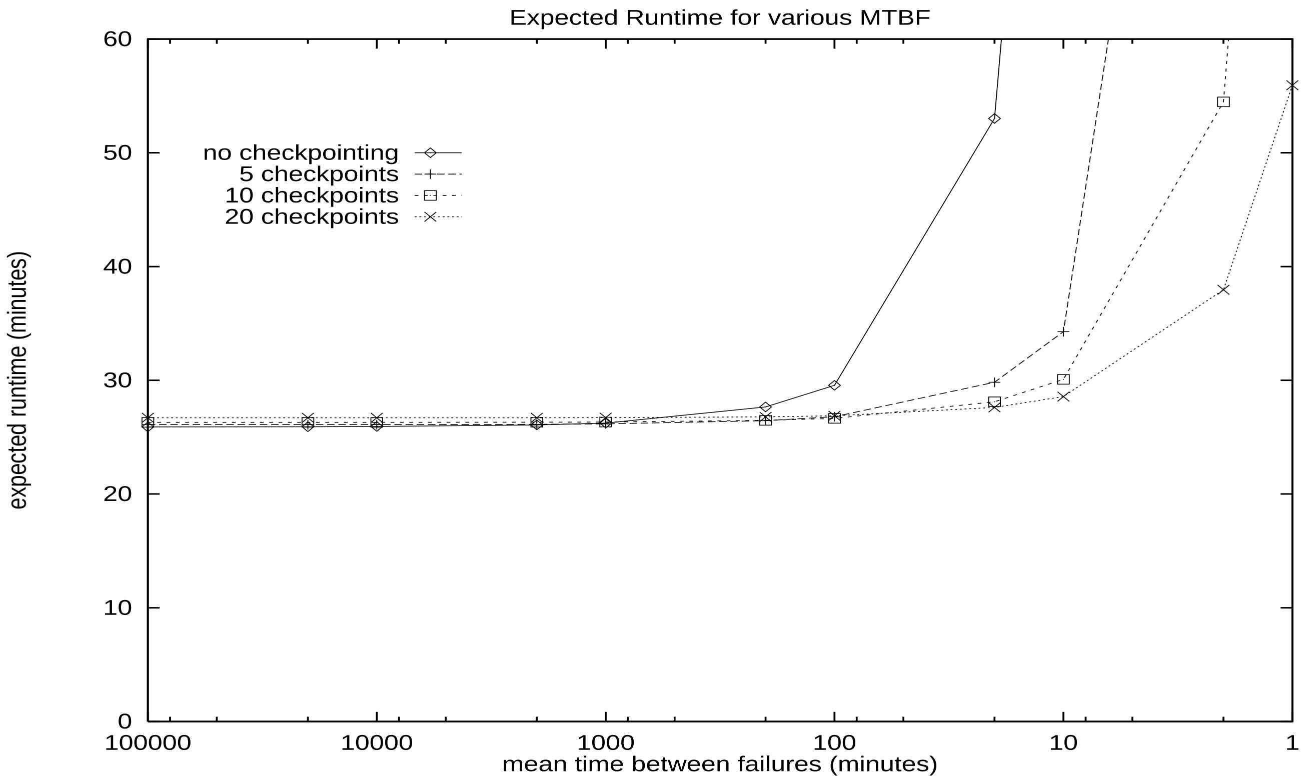

Figure 5: Expected runtime for *md* vs. mean time between failures.

In general, we have seen that high-level checkpointing can be a useful method for implementing fault tolerance in SPMD parallel programming systems such as Dome. High-level fault tolerance sacrifices some user transparency, but provides complete portability for both the checkpointing code and the checkpoints themselves. Our experiments so far have suggested that while only incurring very small checkpointing costs, we can create significant savings in the total expected runtime of a computation in the presence of failures. As we continue our work on reducing the overhead and programmer effort required, while increasing the efficiency and applicability of our system, we believe that our fault tolerance features will be a significant benefit to Dome programmers.

References

[1] Adam Beguelin, Erik Seligman, and Michael Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, May 1994.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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