Recent Advances in Checkpoint/Recovery Systems

Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali and Paul Stodghill*

Cornell University
Department of Computer Science
Ithaca, NY 14853 USA
{bronevet,rohitf,marques,pingali,stodghil}@cs.cornell.edu

Abstract

Checkpoint and Recovery (CPR) systems have many uses in high-performance computing. Because of this, many developers have implemented it, by hand, into their applications. One of the uses of checkpointing is to help mitigate the effects of interruptions in computational service (both planned and unplanned) In fact, some supercomputing centers expect their users to use checkpointing as a matter of policy. And yet, few centers provide fully automatic checkpointing systems for their high-end production machines.

The paper is a status report on our work on the family of C^3 systems for (almost) fully automatic checkpointing for scientific applications. To date, we have shown that our techniques can be used for checkpointing sequential, MPI and OpenMP applications written in C, Fortran, and several other languages. A novel aspect of our work is that we have not built a single checkpointing system, rather, we have developed a methodology and a set of techniques that have enabled us to develop a number of systems, each meeting different design goals and efficiency requirements.

1. Introduction

"Checkpointing" is the process of saving the state of a running application to "stable storage". At some later time, this saved state can be used to "restart" the application at the same point in the computation when the checkpoint was taken. Perhaps the most common use of Checkpoint/Restart (CPR) in high-performance computing (HPC) is to enable applications to continue executing in the presence of certain types of failures. For example, if a machine crashes, then the applications running at the time can be restarted from checkpoints once the machine has been fixed and rebooted.

In addition to hardware failures, CPR can be used to handle other kinds of anomalous events, such as a running application exceeding its time limit or being preempted by a new job with a higher priority. These failures are instances of a class described in the literature as Fail-Stop Faults. There are other, more general classes of faults, such as Byzantine Faults [19]. CPR, by itself, does not provide a complete solution for handling these more general faults, but it is very often a building block that is incorporated into more general solutions.

In addition to failure mitigation and fault tolerance, CPR finds many other uses in HPC. Here are some examples,

Process Migration. For reasons of locality or resource utilization, it is often advantageous to move a running application from one processor to another. This is done by checkpointing the application on the original processor, transmitting the checkpoint to the new processor, and using it to restart the application.

Post-processing. For some applications, particularly those involves time-stepping, the state of the simulated physical system can be deduced from the state at particular execution points in. In this case, a series of checkpoints taken during the execution can be used to visualize the evolution of the physical system over simulated time.

Debugging. CPR can be used to replay a particular program execution. This has been used for isolating race conditions in parallel programs [11]. It has also be used to allow the developer to run the program "backwards", using a technique called reverse debugging [23].

Because it is so useful, it is perhaps not surprising the

*This work was supported by NSF grants #9972853, #0085969, #0090217, #0103723, and #0121401, #0406345, #0509307, and #0509324.

1-4244-0054-6/06/$20.00 ©2006 IEEE

CPR is found frequently in HPC applications and systems. In fact, the Pittsburgh Supercomputer Center (PSC), which provided some of the computational resources used to evaluate this work, expects their users to incorporate CPR functionality as part of their applications. It is the Center's policy [9] that, if a failure occurs during the execution of a user's program, the user will only be refunded for 3 hours of the lost allocation time; users are expected to checkpoint their applications accordingly. However, although there is support for developers who hand-code checkpointing routines into their applications (e.g., [9]), to the best of our knowledge, none of the members of the Teragrid, the Arctic Region Supercomputing Center, or Cornell Theory Center provide fully or semi-automatic CPR systems.

The focus of our research over the last several years has been to develop semi- and fully automatic CPR systems that work in practice. In order to accomplish this goal, we have had to develop new approaches and techniques and evaluate their effectiveness. This paper summaries our accomplishments to date and discusses our ongoing work. In Section 2, we will give a background on the conventional approaches for CPR. In Section 3, we will describe two new approaches for that we have developed or refined. In Section 4, we will describe C^3, our basic system for checkpointing sequential applications. In Section 5 and 6, we will describe how we have extended C^3 to checkpoint distributed memory and shared memory applications, respectively. In Section 7, we will describe how we have extended C^3 to generate transportable checkpoints for heterogeneous computing environments. In Section 8, we will offer some conclusions and discuss our future work. Throughout this paper, we will discuss other work that is related to ours.

2. Approaches to checkpointing

There are two basic approaches to checkpointing: System-Level Checkpointing and Application-Level Checkpointing.

With System-Level Checkpointing (SLC), CPR is implemented in a system that is external to the application. There are many ways to do this. Systems like BLCR [13] and MOSIX [1] are implemented within the operating system. Condor [18] and Libckpt [24] are implemented as user-level runtime systems that are linked, either statically or dynamically, with the application. All of these SLC systems have several features in common. First, they checkpoint and restore the application state by directly copying the raw bytes of the application memory to stable storage and back. Second, because they operate at a low-level and do not require specific knowledge of the application, they provide CPR without requiring any (or many) changes to the application.

With Application-Level Checkpointing (ALC), CPR is implemented directly within the application's source code. That is, the application contains source code that saves and restores critical program variable to and from stable storage. This approach has two advantages for the developer. First, because it works with the application variables and not with a lower-level representation, the checkpointing code can be made portable without too much difficulty. Furthermore, with a little more effort, it is possible to make the checkpointing data portable as well. This enabled CPR within heterogeneous computing environments. Second, the developer is able to write CPR code that only saves the minimum set of variables that are needed to restart the application. In practice, this can result in tremendous reductions in checkpoint sizes (anecdotally, we have found that the savings can range from 90% on some DOE lab codes to million-fold reductions on certain protein folding codes).

We can compare the relative merits of each approach by considering several different characteristics.

Transparency - the degree to which CPR can be added to an application without changing it. Generally, SLC is more transparent than ALC. Systems such as MOSIX and Condor are engineered so that no modifications have to be made to the application in order to incorporate the CPR functionality. On the other hand, a developer that wishes to incorporate ALC into their code must think carefully about what points in the execution are appropriate for checkpointing and which program variables must be saved in order to successfully restart the application. This task can be very time consuming; in fact, it was reported to us that for some of the production applications at the DOE labs the work is the equivalent of 1 FTE.

Portability - the ability to move the CPR system from one platform to another. Portability is very important to most application developers, who are reluctant to tie their applications too closely to a particular platform. Not only might the developer wish to distribute their application to other users at other institutions, but the developer can be certain that their current favorite platform will be obsolete and replaced in several years. When a developer is choosing a CPR solution, they will likely consider how portable the CPR system is. It would not be prudent to choose a CPR system that locks the developer into a particular platform[1].

Another aspect of Portability is whether or not special system administrator privileges are required in order to install and use a CPR system.

Because it does not work with low-level machine or operating system representations of the application, and because it does not require special privileges to operate, ALC is considerably more portable than SLC. Systems like Condor and MOSIX will only run on certain operating system

[1]Recently, the system administrators in Cornell Computer Science had to cancel an operating system upgrade on the department's main Linux cluster when it was discovered that the cluster's production CPR system (MOSIX) would not run on the new version of the operating system.

and compiler versions, while most developers can get their application source code to run on a wide range of platforms without too much difficulty.

Transportability - the ability to use checkpoint data produced on one platform to restart the application on another. Depending upon the requirements, it may be necessary to move checkpoint data between different platforms. This might be necessary, for instance, in the case when an application is migrated between idle workstations within an organization. It might also be necessary in a Dynamic Data-Driven Application System (DDDAS) [12] in which additional computation resources are brought on-line in order to meet an application's changing resource requirements.

Efficiency - the overhead added to the application's execution by the CPR system. Performance is always an important consideration. Although a developer may be willing to tolerate substantial overhead when restarting an application (assuming that this is an infrequent operation), they are unlikely to tolerate a very high cost for periodic checkpointing. In practice, we have found two sources of overhead to be important. The first overhead is incurred in order to maintain information about the execution of the application that would be used if a checkpoint were taken. This overhead, which we call the *checkpoint-free overhead*, is paid whether or not any checkpoints are taken during execution and should be kept as small as possible. The second overhead is the cost of writing checkpoint data to stable storage whenever a checkpoint is taken, which we call the *checkpointing cost*. We have observed that this overhead is proportional to the size of the checkpoint datay.

As we will see, certain ALC techniques tend to incur a checkpoint-free overhead, whereas the SLC techniques generally do not. The checkpointing cost can be reduced by reducing the amount of data in each checkpoint. In ALC, this is done by the developer deciding that certain application variables do not need to be saved, because they are not needed or can be recomputed upon restart. Since SLC systems do not have any knowledge of the application, they rely upon observing low-level details of the application's execution in order to reduce the amount of state saved (e.g., *incremental checkpointing* [26] relies upon setting protection bits on memory page in order to observe which pages are read and written consecutive checkpoints).

Correctness - the ability of a CPR system to ensure that the application produces a correct result. Of all of the properties that we have listed here, correctness is arguable the most difficult to ensure. To illustrate this point, we consider two examples.

First, consider an application that uses the function, `rand()`, from the C library. As part of the implementation of this function, there is a variable, `seed`, that is used to record the seed of the sequence of pseudo random number generated by the function. An SLC system is likely to save the value of the `seed` variable along with the rest of the application state. On restart, the value of this variable will be restored and the application will continue to run. In this case, the developer can expect the function `rand()` to return the same sequence of values, regardless of whether or not the application is restarted between calls. A developer that is incorporating ALC into an application usually does not have direct access to the `seed` variable (this is true on all versions of Linux, Solaris and Windows that we have used). The developer must rely upon other functions to access and restore the `seed` value during checkpointing. Unfortunately, POSIX provides a function for restoring the `seed` value, but not for accessing it. In this case, the developer must decide whether or not their application requires the sequence of pseudo-random values to be preserved across restarts. If so, then the developer will almost certainly have to implement their own, checkpointable, version of the `rand()` function.

The second example involves the MPI communication library. This library maintains a map from virtual processor id's to physical processor id's (e.g., IP addresses). An SLC system might treat this map as part of the application state and save it in the checkpoint. This would be incorrect, as on restart, the set of physical processors allocated to the application might be different. In this situation, it is generally not necessary to save and restore the processor map; an application that accesses only virtual processor id's and not physical processor id's should still function properly even if the mapping is changed on restart. A developer incorporating ALC into an application will be aware of this and will not attempt to access or restore the map.

As these two examples illustrate, neither SLC nor ALC is a clear winner with respect to correctness.

3. Our Approach to Checkpointing

As part of this project, we have used two novel approaches to checkpointing that build upon the basic SLC and ALC approaches. The new approaches have enabled us to develop entire new classes of checkpointing systems. In this section, we discuss these two approaches and describe the systems that we have built using these approaches in the subsequent sections.

3.1. Mixed-Level Checkpointing

It should be clear from the preceding discussion that neither SLC nor ALC is always the better solution. On the one hand, SLC is generally more transparent than ALC. On the other, ALC can be made portable and transportable more easily. Efficiency and correctness are difficult issues for both approaches.

We have developed a new approach to checkpointing that we call *Mixed-Level Checkpointing (MLC)*. This approach combines aspects of both SLC and ALC in order to develop CPR systems that are able provide strength for each of the five properties discussed above. Using this new approach, we have built not a single CPR system, but a number of CPR systems, each with different design goals and performance characteristics. Although the idea of developing hybrid SLC/ALC systems may appear obvious, what is not at all obvious is how to go about doing this. The great difficulty in building MLC systems is separating the state of an application into logically consistent sets that can the be checkpointed using either SLC or ALC approaches.

Let us consider the two examples that were used in the discussion of correctness above. Suppose that we wished to build an MLC system that used ALC for saving the global variables of the application and SLC for saving the state of the `rand()` library, including the `seed` variables. In order to be able to do this, it is necessary to separate the application state into two sets, the application global variables and the variables used by the `rand()` library. Our solution to this problem relies on the fact that the compiler and linker on most systems ensure that program variables are laid out in memory in a certain way. In this case, it is possible to use a combination of special compiler-generated global variables and special linker commands in order to isolate the two sets of variables into two different and identifiable regions of memory. Suppose that we wished to use SLC for saving the application variables and we wished to never save the internal state of the MPI library. In this case, the same techniques can be also be used. A complete description of the set of the MLC mechanisms that we have developed for isolating various sets of state and for incorporating different combinations of ALC and SLC techniques into a single CPR system can be found in [20].

[20] also discusses how different combinations of ALC and SLC techniques can be used to develop many different CPR systems with varying feature and performance characteristics. This is critical when we start to consider the correctness of a CPR system in a systematic manner. This is because "correctness" is a concept that must be defined on a per-application basis. Consider two different applications that called the system's time function during their execution. When checkpointing these applications, how should the "state" of the time function be handled? The answer depends upon the application. One application might generate log messages that contain the wall clock time, while the other might use two clock readings to determine the performance of a computational kernel. Different checkpointing solutions are required to handle these two different application correctly. The ability of our MLC techniques to generate, not a single CPR system, but a wide range of CPR systems allows us to generate different CPR systems based upon different correctness requirements and is one of this projects greatest contributions to checkpointing.

3.2. Automatic Application-Level Checkpointing

One of the disadvantages of ALC is that it requires the developer to add the CPR functionality to their code by hand. An obvious improvement would be to automate this process, which we call *Automatic Application-Level Checkpointing (AALC)*. The basic idea is for a program transformation tool or pre-compiler to analyze the application source code and determine what program variables must be saved at each checkpoint and to add the appropriate code to the source code to write checkpoints and to restart the application from these checkpoints.

We did not develop the basic idea or techniques for AALC; our work builds upon systems such as Porch [27] and APrIL [16]. Nevertheless, our works improves upon these in a number of ways. Our techniques are described briefly in [4] and in much greater detail in [20].

A novel aspect of our approach of combining MLC with AALC is that we are able to develop new compiler analyses and optimizations to reduce the amount of data being saved at each checkpoint. In our approach, a static analysis is done at compile time to compute information that can be fed to the runtime system to reduce the checkpointing overhead. In [21] we describe how we have added functions to our heap implementation that allows heap objects to be partitioned into "colors". There are additional functions for assigning checkpointing policies to each color (e.g., "Never save this color" or "Save this color only once"). We are currently developing the compiler analyses that will use these API. Other work that has been done on compiler-directed memory exclusion can be found in [2] and [25].

4. Checkpointing sequential applications

The first CPR system that we developed based upon on our new MLC and AALC techniques was C^3, which initially provided checkpointing for sequential applications written in C. The first version, C^3/LE used the AALC described in [4] and [20] for checkpointing the global and local variables of the application code and the execution context. A custom memory manager was provided for checkpointing heap objects using SLC techniques. The two sets of techniques were combined using the MLC techniques that we developed.

For most of the standard benchmark codes, we found the overhead of using our system to be quite small (usually between 5% and 10%), but in a few cases we found the overhead to be much larger. The reason turned out to be that the AALC code that C^3 inserted into the application was

preventing the native compiler from doing as good a job at optimization. In addition, Prof. McKee's group at Cornell started using C^3 to support their work in using CPR for dramatically speeding up hardware simulation [31]. For their application, changing the source code of the applications was deemed to be extremely undesirable, as it tended to change the behavior being simulated.

For these reasons, we developed a second version of our CPR system, C^3/ME, that did not use AALC, but instead relied upon a number of different SLC techniques combined using MLC. In addition to avoiding the performance anomalies that we observed with C^3/LE, because it did not require parsing source code, we were able to extend C^3/ME to handle languages other than C (e.g., FORTRAN 77 and some parts of C++).

Because it requires making only minor changes to the application (i.e., adding annotations to indicate program locations at which to checkpoint), C^3 provides an almost completely transparent CPR solution. Furthermore, we have demonstrated its correctness and efficiency on a number of different computing platforms, including Linux, Solaris, Windows, AIX and Tru64 UNIX. A detailed performance evaluation of the two versions of C^3 can be found in [20].

5. Checkpointing distributed-memory applications

Our next challenge was to extend our basic sequential checkpointing system to handle message-passing, distributed memory programs, such as those written using MPI [17]. There are two key difficulties with checkpointing MPI applications. The first is the presence of "hidden" state within the MPI library, including such things as communicators, user-defined data-types, and handles for asynchronous communication objects. The second difficulty is the presence of in-flight messages when checkpoints are taken. Hidden library objects and in-flight messages both represent state that is inaccessible to the application but that must be checkpointed and restored for correctness.

One approach that has been used in the past is to build CPR functions directly into the MPI library. When it is time to checkpoint, special functions are called to checkpoint the hidden state and flush the in-flight messages. This approach requires a tight integration between the CPR system and the MPI implementation. An example of such a system is CoCheck [30], which integrates the Condor CPR system with the MPICH MPI library. Recently CPR hooks have been added to LAM MPI [8, 29] in order to enable its integration with CPR systems. The disadvantage of this approach is that there are many systems on which it is not possible to change the production MPI implementation. This may be because the implementation is proprietary and its source code is not available, or it could be because the system administrators do not want users to deploy modified versions of such a critical component on production systems. Because of these reasons, we were not in a position to deploy modified versions of MPI on any of the large clusters that we used for this project.

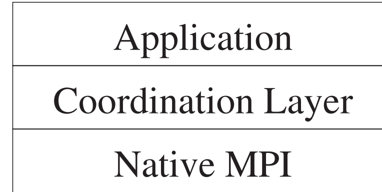

Figure 1. Architecture of C^3 for MPI

As a result of these limitations, we investigated techniques that enable our C^3 systems to work in conjunction with whatever native implementation of MPI is found on a machine. In particular, we have designed a thin coordination layer that sits between the application and the native MPI implementations, as shown in Figure 1. This coordination layer intercepts all MPI calls made by the application and performs certain tasks that enable the communication portion of the application state to be checkpointed and restored. Because of this design, our system is able to work with any standard-compliant MPI implementation. To date, we have demonstrated it using LAM, MPI/Pro, and several variants of MPICH, on a wide range of machines.

To checkpoint the hidden state in the MPI library, the coordination layer records information about the opaque communication objects, such as communicators and data-types, when they are created. This information is saved to stable storage along with the checkpoint. When an application is restarted, the coordination layer uses this information to recreate these objects within the MPI library.

Handling in-flight messages is more difficult. One approach is to ban them. That is, we could rely upon the application developer to synchronize the checkpoints on different processors using, for example, a barrier. The developer must also ensure that when checkpoints are taken there are no messages in-flight between processors. This approach, which is called *Blocking Coordinated Checkpointing* [14], is sufficient for many existing codes that have been written in a bulk-synchronous, or BSP, manner.

However, there are emerging classes of parallel applications, such as the parallel mesh generators described in [22], which askew the BSP model in favor of a more asynchronous and self-synchronizing approach to communication. In these codes, there are few, if any, points in the execution when the developer can be certain that there are no messages in flight and require a different class of protocols, called *Non-blocking Coordinated Checkpointing* [14]. Chandy-Lamport [10] is perhaps the most well known ex-

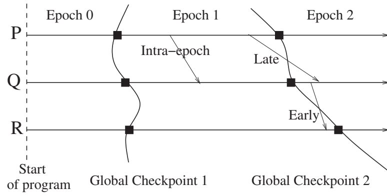

Figure 2. Epochs and messages in-flight

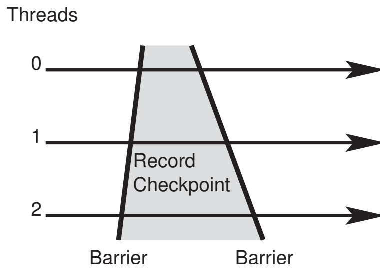

Figure 3. Checkpointing a shared memory application

ample of a non-blocking coordinated checkpointing protocol. Unfortunately, it requires the use of preemptive checkpointing, which is not possible with our ALC techniques.

For these more general classes of applications, we have developed novel non-blocking coordination protocols that enable the C^3 system to non-preemptively checkpoint virtually any correct MPI application. Our protocol for handling point-to-point communication is described in [3] and illustrated using Figure 2. Our protocol works by dividing the execution of the application into *epochs*. Epochs are divided by *recovery lines*; a recovery line is a set of checkpoints that can be used to restart the application. When one processor takes a checkpoint, our protocol guarantees that it will be matched with checkpoints from all other processors to form a recovery line, or it is discarded.

As shown in Figure 2, messages can be classified depending upon how they cross recovery lines. Figure 2 illustrates three kinds, *early*, *late*, and *intra-epoch*. By piggybacking certain information on each message, our protocol is able to classify all messages, and to ensure that on recovery the appropriate actions are taken for each in-flight message (e.g., late messages are read from a log by their receiver, early message are suppressed by their senders).

In [5], we show how the basic concepts of the point-to-point protocol can be extended to define protocols for handling the collective communication constructs of MPI. In [28], we present experimental results on up to 1024 processors that show that our protocols deliver scalable performance. The overhead that we observed was usually between 5% and 10%, which are acceptable overheads in practice.

6. Checkpointing shared-memory applications

Extending C^3 for shared memory programs is more challenging. For MPI, the points at which communication and synchronization occur are explicit; they only occurs through calls to the MPI library. In the case of shared memory programs, these interactions can be implicit. In fact, without extensive compiler or hardware support, it is not clear that it is possible to develop a non-blocking coordination protocol for shared memory programs. For the present, we have focused on developing blocking coordination protocols. Because our protocols are blocking, we are able to implement them independently from the underlying shared memory implementation. Furthermore, we believe that blocking protocols are less onerous for shared memory applications than distributed memory because the largest shared memory machines have several orders of magnitude fewer processors than the largest distributed memory machines. However, we continue to investigate non-blocking solutions.

Our protocol [6, 7] for shared-memory applications is illustrated in Figure 3. The first step is for all threads to execute a barrier instruction. Once this has been done, then all threads write their private data to the checkpoint file and coordinate writing the global data as well. Next, all threads execute a second barrier instruction. Once this is completed, then all threads resume executing application instructions.

This protocol is extremely simple in principle, but can be extremely difficult to implement in practice. We are currently working on describing how the basic protocol can be implemented for OpenMP applications. This requires addressing a number of interesting issues. First, suppose that when the checkpoint protocol is initiated, some of the threads were already blocked at a barrier. In [6], we describe a technique whereby a sequence of barrier instructions are executed both before and after the protocol in order to ensure that blocked threads execute the checkpoint protocol and then are returned to a blocked state.

OpenMP presents another challenge in that many of its synchronization mechanisms are specified as programming language constructs. This makes it necessary to incorporate AALC techniques in order to checkpoint OpenMP codes. For instance, if a thread is blocked at the start of a critical section when a checkpoint is taken, then the only way to release the thread to run the protocol is for the thread that is currently executing the critical section to exit it.

7. Checkpointing in a heterogeneous computing environment

All of the C^3 systems that we described above meet four of the five criteria for checkpointing systems, namely Transparency, Portability, Efficiency and Correctness. None of them, however, provides Transportability, that is, the ability

to use checkpoint data generated on one platform to restart an application on a different platform. For example, one might wish to checkpoint an application on the Lemieux cluster at PSC, which has over 3000 64-bit Alpha processors, Compaq compilers, a vendor-supplied MPI, and runs Tru64 UNIX, transport the checkpoint to the Cornell Theory Center (CTC) and restart the application on one of the CTC's Velocity clusters, which has 256 32-bit Pentium processors, Intel compilers, MPI/Pro, and run Windows Server 2000. In [15], we describes how our homogeneous checkpointing techniques can be extended to provide heterogeneous checkpointing for this scenarios.

There are a number is problems that must be solved in order to provide transportable checkpointing for this scenario. Some are relatively simple to address. For instance, the fact that the two clusters have different versions of MPI is of no consequence, because our coordination layer is written for a generic MPI implementation and does not exploit any implementation specific properties.

The fact that both clusters use different processor architectures, compilers, and operating systems means that we cannot naively save the application state as binary data on one processor and expect to use it directly on their other. The checkpoint data generated from Lemieux must be translated in order for it to be used on Velocity. In order to do this translation, each object in the checkpoint data must be assigned a unique type. Assuming that this assignment is done, then translating objects between machine representations is possible. However, applications written in the C programming language do not provide unique type assignments to memory objects. Heap objects in C, for example, are treated as untyped regions of memory that the developer is free to use in any way that they see fit. This means that type information is not available when heap objects are checkpointed. Furthermore, C provides syntactic constructs that allow the developer to access objects with a different type than their declared type. It is permissible, for instance to treat objects as arrays of characters in order to access their byte representations.

Our solution to this problem is to impose a stronger type discipline on C applications. In [15], we describe a combination of static and dynamic type checking techniques that our system uses to ensure that unique types are assigned to each object that is checkpointed. We also describe how information is saved about how each type is represented on the machine generating the checkpoint. This information, along with the type assignments, is used to translate the objects to a new representation when the checkpoint is transported to a different machine.

Another problem we address in [15] is the fact that the checkpointing and restoring machines may have a different number of processors. For instance, if we checkpoint an application running on 256 processors of Lemieux, how do we restart it on 64 processors of Velocity? The technique that we use is *over-decomposition*. When the application is started on 256 processors of Lemieux, we start the application running on, say, 512 virtual processors. In order to do this, our system must map 2 virtual processors to each physical processor of Lemieux. When checkpointing, the state of the 512 virtual processors is saved. This checkpoint data is transported to Velocity, translated, and used to restart the application on 64 processors. In order to do this, our system must map 8 virtual processors to each physical processor of Velocity. Our system uses a combination of program transformations and runtime support to map each virtual processor onto a thread running on one of the physical processors. Our runtime system enables threads to communicate through MPI using their virtual processor ranks instead of the physical processor ranks.

8. Conclusions

At present, we are currently working in two major directions. First, we are refining our current implementations so that they can be released and used in production environments. As part of this effort, we are revisiting some of our basic assumptions about CPR systems. For instance, in our first implementation of C^3, we placed a premium on portability. We are currently investigating whether or not, for instance, there are certain ways that we can trade a certain amount of portability for an even higher level of efficiency.

Second, we are continuing our research into compiler analyses and optimizations for automatically reducing the amount of state saved. We have conducted a careful analysis of a number of application codes and we have been able to establish reductions in checkpoint size of 50%-90%. Our hope is that a program transformation system can achieve most or all of these savings with as little help from the developer as possible.

In this paper, we have described the family of C^3 systems for (almost) fully automatic CPR for scientific applications. To date, we have shown that our basic techniques can be used to provide CPR for sequential, MPI and OpenMP applications written in C, Fortran, and several other languages. One of the novel aspects of our work is that we have not built a single checkpointing system, rather, we have developed a methodology and a set of techniques that have enabled us to develop a number of systems, each meeting different design goals and efficiency requirements. This is because, in contrast to previous work that uses either System-Level Checkpointing (SLC) or Application-Level Checkpointing techniques (ALC), ours is based upon Mixed-Level Checkpointing (MLC), which is a new concept, and Automatic Application-Level Checkpointing (AALC), to which we have added many refinements.

References

[1] A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*. Number 672 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[2] M. Beck, J. S. Plank, and G. Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, Dept. of Computer Science, University of Tennessee, 1994. Also available as `http://citeseer.nj.nec.com/173887.html`.

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

[4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C^3: A system for automating application-level checkpointing of MPI programs. In *The 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, 2003.

[5] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault-tolerant mpi system. In *International Conference on Supercomputing (ICS)*, June 2003.

[6] G. Bronevetsky, M. Schulz, P. Szwed, D. Marques, and K. Pingali. Application-level checkpointing for shared memory programs. In *Conference on Application Support for Programming Languages and Operating Systems*, 2004.

[7] G. Bronevetsky, M. Schulz, P. Szwed, D. Marques, and K. Pingali. Checkpointing shared memory programs at the application-level. In *European Workshop on OpenMP*, 2004.

[8] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[9] P. S. Center. Lemieux. Available at `http://www.psc.edu/machines/tcs/lemieux.html`.

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

[11] J.-D. Choi and J. M. Stone. Balancing runtime and replay costs in a trace-and-replay system. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Dec. 1991. published in ACM SIGPLAN Notices, 26(12):26–35.

[12] C. Douglas, A. Deshmukh, et al. Report from the March 8-10, 2000 NSF sponsored workshop on Dynamic Data Driven Application Systems. Accessed February 8, 2003.

[13] J. Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. http://www.nersc.gov/research/FTG/checkpoint/reports.html.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375 –408, Sept. 2002.

[15] R. Fernandes, K. Pingali, and P. Stodghill. Mobile mpi programs in computational grids. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2006. To appear.

[16] A. Ferrari, S. J. Chapin, and A. S. Grimshaw. Heterogeneous Process State Capture and Recovery through Process Introspection. In *Cluster Computing*, pages 63–73, 2000.

[17] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[18] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison, 1997.

[19] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, first edition, 1996.

[20] D. Marques. *Automatic Application-Level Checkpointing for High Performance Computing Systems*. PhD thesis, Cornell University, Jan. 2006.

[21] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghill. Optimizing checkpoint sizes in the c^3 system. In *NSFNGS 2005 Workshop*, 2005. Held in conjunction with, IPDPS 2005.

[22] D. Nave, N. Chrisochoides, and P. Chew. Guaranteed–quality parallel Delaunay refinement for restricted polyhedral domains. *Computational Geometry: Theory and Applications*, 28(2–3):191–215, June 2004.

[23] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124 – 129, Madison, Wisconsin, 1988.

[24] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under UNIX. In *USENIX Winter*, pages 213–224, 1995.

[25] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: optimizing the performance of checkpointing systems. *Software Practice and Experience*, 29(2):125–142, 1999. Also available at `http://citeseer.nj.nec.com/plank96memory.html`.

[26] J. S. Plank, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, Aug. 1995. Also available at `http://www.cs.utk.edu/~plank/plank/papers/CS-95-302.ps.Z`.

[27] B. Ramkumar and V. Strumpen. Portable Checkpointing for Heterogenous Architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[28] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Supercomputing '04*, 2004.

[29] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[30] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[31] P. Szwed, D. Marques, R. Buels, S. McKee, and M. Schulz. SimSnap: Fast-forwarding via native execution and application-level checkpointing. In *Interact-8: Workshop on the Interaction between Compilers and Computer Architectures*, 2004.