Application-Level Checkpointing Techniques for Parallel Programs*

John Paul Walters[1] and Vipin Chaudhary[2]

[1] Institute for Scientific Computing
Wayne State University
jwalters@wayne.edu
[2] Department of Computer Science and Engineering
University at Buffalo, The State University of New York
vipin@buffalo.edu

Abstract. In its simplest form, checkpointing is the act of saving a program's computation state in a form external to the running program, e.g. the computation state is saved to a filesystem. The checkpoint files can then be used to resume computation upon failure of the original process(s), hopefully with minimal loss of computing work. A checkpoint can be taken using a variety of techniques in every level of the system, from utilizing special hardware/architectural checkpointing features through modification of the user's source code. This survey will discuss the various techniques used in application-level checkpointing, with special attention being paid to techniques for checkpointing parallel and distributed applications.

1 Introduction

When programmers seek to write code resilient to failures, the typical course of action is to resort to application checkpointing. Application checkpointing is the act of saving the state of a computation such that, in the event of failure, it can be recovered with only minimal loss of computation. This is especially useful in areas such as computational biology where it is not unusual for an application to run for many weeks before completion [1]. In such cases it is possible for the program's running time to exceed the hardware's failure rate. If the application cannot be recovered from some intermediate point in the computation, it is reasonable to expect that the application may never finish. A high-level overview of the necessary components of checkpointing could be written as:

1. Interrupt the computation.
2. Save the address space to a file.
3. Save the register set to a file.

* This research was supported in part by NSF IGERT grant 9987598 and the Institute for Scientific Computing at Wayne State University.

S. Madria et al. (Eds.): ICDCIT 2006, LNCS 4317, pp. 221–234, 2006.

Other attributes could also be saved, including sockets, open files, and pipes. But the items listed above suffice to accurately restore a computation.

The problem is compounded when considering clusters of machines, all of which fail independently at some given rate. If the application makes use of the entire cluster, then the failure of a single machine will halt progress of the entire application. In these cases programmers would like to make use of application checkpointing schemes in the distributed environment, but the problem can no longer be solved by simply taking a snapshot of each process as there may be messages in transit that none of the individual processes would lay claim to.

Consider also the case of grid computing. In this case, all of the complexities of computing in the distributed manner apply (messages in transit, synchronizing multiple processes) but new problems arise as well. In a grid environment individual machines may be under the control of different administrative domains. Indeed the identity of any particular node responsible for running a programmer's code may be unknown to the programmer. In such cases, checkpoints cannot simply be saved to local filesystems, but must be funneled to a known location. Furthermore, the heterogeneous nature of the grid makes it exceedingly possible that machines of different architectures are allocated to the programmer upon resuming the application. One solution to this problem would be to mandate a certain architecture for the application. While such a solution would work, a better solution would handle checkpoint resumption on heterogeneous architectures.

To address the problems above, a series of checkpoint techniques have been developed that provide programmers with varying degrees of checkpointing transparency. The four major categories are as follows [2,3]:

1 Hardware-level, additional hardware is incorporated into the processor to save state [3].
2 Kernel-level, the operating system is primarily responsible for checkpointing running programs [4,5,6].
3 User-level, a checkpointing library is linked into a program that will be responsible for checkpointing the program independent of the programmer [7,8].
4 Application-level, the checkpointing code is inserted directly into the application by a programmer/preprocessor.

This survey will present the major techniques used in application-level checkpointing with special attention being paid to distributed/grid applications. In figure 1, a taxonomy of common application-level checkpointing schemes is presented, including a classification of the specific implementations discussed in this survey.

A related problem is the issue of process and thread migration [9,10,11,2]. While this survey does not specifically address the issue of process migration, it is worth noting that many of the challenges associated with application checkpointing arise in the case of process migration as the two problems are nearly identical in nature.

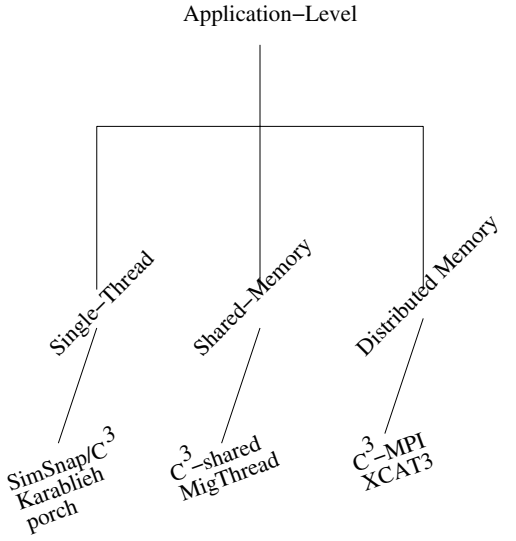

Fig. 1. A taxonomy of application-level checkpointing schemes

2 Application-Level Checkpointing

Application-level checkpointing represents the highest degree of abstraction from the process itself. Unlike the more transparent techniques of kernel-level checkpointing and user-level checkpointing, application-level checkpointing requires more programmer interaction. It would seem then that application-level schemes are at a disadvantage when compared to their user-level and kernel-level counterparts. But application-level checkpointing carries with it several distinct advantages over the lower-level techniques, including its overall portability.

2.1 Introduction

The essence of application-level checkpointing is to place the checkpointing burden on the application itself, rather than the operating system or a set of user-level libraries that are linked into the program. In order to accomplish the self-checkpointing, the program source must be modified to support checkpointing. Unlike other techniques (such as user-level checkpointing), application-level checkpointing seeks to abstract the application state to the application-level. Thus, the process state is not saved through low-level system calls. Instead, compiler assistance (typically in the form of a pre-processor) is used in a technique called "code instrumentation" [12,1,13,14,15,16,17,18,19].

Ideally, the use of application-level checkpointing would not require additional effort from the programmer. That is, the programmer should not be held responsible for adding the checkpointing code himself. To avoid this, the pre-processor scans the original code provided by the programmer, inserts the necessary checkpointing constructs, and outputs the checkpoint-enabled code. The new code

should function no different than the original, with the exception of the addition of the checkpointing functionality.

As mentioned above the application-level code must represent the low-level program status (program counter, etc.) in terms of higher level constructs. For instance, application-level stacks are implemented to keep track of a program's place when nested inside function calls. Each time the application enters a function, the function ID (another application-level construct) is pushed onto the checkpointing stack. Similarly, when an application leaves a function, the function ID is removed from the checkpointing stack.

Since the application does not make use of the low-level representation of the process, checkpointing code must be inserted such that a program simply executes the checkpointing code at appropriate times during its execution. Of course, this means that the code can no longer be checkpointing at arbitrary points (via interruption). Furthermore, when the application is restored the application-level scheme cannot simply restore the program-counter to return the application to the appropriate point. Instead, a series of jump statements are used to allow the application to simply initialize variables but skip the checkpointed computations in order to recover from a failure.

While the above may seem to be a disadvantage to the application-level scheme, such a technique allows for three distinct advantages: language independence/portability, checkpoint size, and checkpoint heterogeneity. We now consider each of these in turn.

At the user-level, the checkpointing schemes rely heavily on system calls provided by the operating system in order to gain access to low-level process information. Furthermore, these system calls are not necessarily available in all languages. Most user-level techniques appear to assume that the "C" language is the only one used by programmers, including the area of scientific computing. This doesn't mean that user-level techniques cannot be used in different languages, but that the techniques must be adapted depending on the environment. Application-level checkpointing schemes, on the other hand, have the potential to be implemented independent of the language provided that the basic language constructs are present.

The checkpoint size is also of great concern when checkpointing large applications. Using standard user-level checkpointing techniques or kernel-level techniques typically save nearly the entire process state including data that need not be saved in order to recover the application. For example, in [1] the authors note that in "protein-folding applications on the IBM Blue Gene machine, an application-level checkpoint is a few megabytes in size whereas a full system-level checkpoint is a few terabytes." Of course the example just presented is rather drastic, but the advantage remains clear - application-level checkpointing has the capacity to dramatically reduce checkpoint size. The advantage to smaller checkpoint sizes is even more clear when checkpointing grid-based applications.

The third major advantage is that of heterogeneity. This too is particularly advantageous in grid-enabled applications. In such an environment, one may not have access to sufficient homogeneous resources. In fact, one may not even have

access to the same grid resources in subsequent application runs. For this reason, heterogeneous checkpointing becomes a distinct advantage. It potentially allows a programmer to take advantage of many more resources that would otherwise go unused. But the real advantage occurs when a node fails. In such a case it may happen that the only available nodes to replace the failed one are heterogeneous to the failed node. In such a case, using a heterogeneous checkpointing scheme, a new node can be selected without regard for its system software or architecture. Furthermore, application-level schemes allow a larger portion of the application to be checkpointed and restored to heterogeneous systems, including pointers.

Of course, there are disadvantages to the application-level technique. For example, applications can only be checkpointed if their source code is available to the pre-processor. Another shortcoming to application-level checkpointing is the difficulty in representing a running program's state. As discussed previously, an application-level technique cannot depend on low-level system calls to discern a running program's state. Similarly, multi-threaded applications must take special care in order to ensure thread synchrony, given that multiple threads will not reach a checkpoint simultaneously.

The final major shortcoming of the application-level technique relates to the issue of heterogeneity. While the application-level scheme may naturally lend itself to portability, one major problem is the manner in which data should be represented. Since different architectures represent data types in different sizes (not to mention different endianness), a technique to convert data on one architecture to meaningful data on another is needed. One standard solution is to use a machine independent technique, the classic example being XDR (External Data Representation) [20]. Some newer application-level schemes have used an XML-based format as well [21]. The major shortcoming to using an intermediate format technique is that all data must be converted twice, once during the checkpoint and again during the restore. Depending on the amount of data that needs converting as well as the frequency at which checkpoints are taking, this extra conversion could be significant. Another technique is to appeal to the lowest precision in the group [22] or to save all checkpoint data in a precision that is higher than that of any of the machines within the group. Both of these techniques suffer from the disadvantage that a conversion is required, even if the checkpoint is being restored to the machine on which it was taken. In addition, the technique of saving in the lowest precision of the group requires knowledge of the group members before checkpointing takes place. In many situtations, particularly grid scenarios, this information may be unavailable. And saving in a precision that is higher than any member of the group not only requires knowledge of the group, but would also lead to inflated checkpoint sizes.

A third technique to data conversion seeks to eliminate the conversion step altogether if at all possible. Ine one such technique, called "receiver makes right," the originator of the data simply checkpoints the data in its own precision. This technique has been used in [14] as well as in PVM [23]. In this case, the receiver is charged with ensuring that the data conversion takes place when necessary. In many cases, such as when a checkpoint is restored to the same machine on

which it was taken, the data conversion can be skipped altogether. In the case of large checkpoints, the time savings could be significant. Should the receiver's architecture differ such that data conversion is necessary, the checkpoint data can be converted.

At the outset, the "receiver makes right" scheme seems like the logical choice. However, data conversion issues arise in the case of architecture differences. For example, a 32 bit data type can be adapted to fit in to a 64 bit data type on another architecture with relative ease. But how to handle the case of adapting a 64 bit data type to a 32 bit data type is non-trivial. In such a case, the possibility exists that the conversion simply cannot proceed as the resulting inaccuracy would be unacceptable.

We next consider three types of application-level checkpointing techniques: single-threaded, multi-threaded, and cluster/grid-enabled application-level checkpointing. Each of these categories carries with it its own difficulties, with each building upon the previous. We begin with single-threaded checkpointing.

2.2 Single-Threaded Application-Level Checkpointing

In [17] a technique similar to PORCH [22] is discussed. The goal is to provide a heterogeneous checkpointing scheme capable of checkpointing "real" programs, including pointers and unions, both of which are known to cause portability problems. They make use of a pre-processor to transform a programmer's code into a functionally similar, but checkpointable, program.

To address the data conversion issues, [17] uses a technique similar to the "receiver makes right" method. Data is always saved in the accuracy of the machine on which the checkpoint is performed. They force the restoring process to first determine its own accuracy and then perform any necessary conversion. Each process can determine its accuracy at runtime, which they claim makes their technique portable to any architecture. The accuracy of a particular architecture is determined by exploiting properties of computer arithmetic developed by [24].

There are several different techniques used to save the state of variables in [17]. Global variables are saved using a series of macros that are generated by the pre-processor. To handle stack variables, the pre-processor encapsulates all local (stack) variables within a particular scope inside a structure. Just as in the case of the global variables, a series of macros are generated in order to facilitate saving the particular variable, which in this case is a structure of variables.

The program itself will likely make calls to additional functions. In order to facilitate rebuilding the execution stack a data structure called the "state stack" is used. Each element of the "state stack" contains three fields. The first is a pointer to the structure containing the local variables. The second is a pointer to a function (generated by the pre-processor) that is capable of saving the state of the local variables. The final field is a label to the callout position of the current scope. The callout of a function is simply the point at which a new function is called from the current scope [17].

In order to facilitate the restoring of an application, an application-level abstraction of the program counter is needed. This is accomplished by making use

of the labels described above. Before each function call (with the exception of checkpoints) a label is inserted. A label is also inserted after a checkpoint [17]. This label is used in the nodes that are pushed onto the "state stack" described above. Such a technique allows a restored application to skip unnecessary computation. Upon restoration, the function simply rebuilds its state stack. This is done by restoring the local variables on each node of the state stack and then skipping directly to the next function call (using the labels). Function calls are then entered in the correct order and their variables restored accordingly.

One common difficulty in application checkpointing, particularly heterogeneous checkpointing, is how to handle pointers and other dynamically allocated data. Karablieh, Bazzi, and Hicks solve this problem by the introduction of a memory refractor, which essentially introduces an additional level of indirection to achieve portability [17]. The memory refractor itself is an array of data structures and is divided into three parts including the global variables, stack, and heap variables [17]. All variables, including non-pointers are kept in the memory refractor. In doing so, the authors are able to effectively abstract the specific memory location from the particular variable by introducing an index for each variable. Pointers then point to the memory refractor entry for a variable, rather than a specific memory location. This, according to [17] allows for portable pointers as well as rebuilding pointers upon recovery. Karablieh, Bazzi and Hicks acknowledge that this is not necessarily the most efficient scheme, but suggest that the portability and heterogeneity of their technique outweighs its inefficiency.

In [18] a novel use for application-level checkpointing is discussed. Rather than exploiting application-level checkpointing for its fault-tolerance and load-balancing characteristics, Szwed et al. describe its use in fast-forwarding applications towards architectural simulation. One can consider fast-forwarding, in their sense, to be roughly equivalent checkpointing/restarting. The difference in this case is that application is restored to a simulator from natively executing code. This acts roughly as an accelerator where researchers can use native execution to "fast-forward" to the more interesting code segments. At a certain point in the code a checkpoint is then taken which is then restored to a simulator. The simulator then continues the computation without modification to the simulator itself. According to [18], such a technique is useful for performing time-consuming cycle accurate simulations on only a particular code segment of interest.

To perform the checkpointing and migration necessary for the simulator, Szwed et al. make use of the Cornell Checkpoint Compiler (C^3) [25,26]. C^3 consists of two components: a source-to-source compiler (pre-processor) and a runtime library. As would be expected, the pre-processor is used to convert the source code of an application into a semantically equivalent version, capable of self-checkpointing [18]. The runtime library includes checkpointable versions of several key "C" library functions.

The C^3 system also makes use of a "position stack" that acts similar to the "state stack" used in [17]. In [18], a label is still inserted after every checkpoint

and before every function that might lead to a checkpoint. The authors perform a call-graph analysis in order to determine which function calls could lead to a checkpoint.

The most unique aspect of C^3 is the method used to restore variables and pointers. C^3 depends on variables being restored to their original memory locations in order to properly restore pointers. To do this, C^3 contains its own versions of the standard "C" memory functions. The most notable of these is the "memory allocator." In this case the memory allocator not only ensures that data is restored to its original address, but also manages heap data in the memory pool. The memory pool is what allows the allocator to restore pointers. Upon restart, the allocator requests from the operating system that the exact same pool of memory be allocated to the process. In doing so, variables can be simply copied from the checkpoint file back into memory without requiring any additional pointer indirection such as in [17].

2.3 Shared-Memory Application-Level Checkpointing

Of course, application-level checkpointing is not limited to single-threaded applications. Indeed there are many different application-level checkpointing schemes targeted at shared memory applications. Here we will discuss two such techniques, a variation on the C^3 scheme, and *MigThread*. These techniques represent two different approaches to the application-level checkpointing scheme as well as different assumptions related to the underlying architectures.

We begin with *MigThread* [27]. In this case, Checkpoints are inserted by the programmer, and it is the programmer's responsibility to ensure program correctness when executing the checkpoint. That is, in order to ensure that an accurate global state is checkpointed, the programmer should enclose the checkpoint in any necessary barriers. The standard technique of using a preprocessor as well as a runtime support module is used in *MigThread* as well.

What's unique about *MigThread* is its heterogeneity and its ability to checkpoint/restart individual threads on remote (possibly heterogeneous) machines. The key to its heterogeneity is its data conversion technique, dubbed "course-grain tagged receiver makes right" (CGT-RMR). This technique allows the sender (or checkpoint initiator) to save the checkpoint data in its own native format. Any conversion that needs to be done will be performed by the receiver upon restarting the checkpoint. This reduces the number of conversions required, possibly to zero if the architecture of the restored checkpoint is the same as when the checkpoint was taken. Data is identified by a series of tags that are generated automatically by the pre-processor and inserted into the user's source code.

One advantage to the technique used by *MigThread* is that is can handle both pointers and dynamic memory without requiring the use of a memory refractor as in [17]. Variables are collected into two different structures, depending on whether they're pointers or non-pointers. This is done at both a global level and at the function level. Upon restoring a checkpoint, the pointers within the pointer structure can be traced and updated according to their new memory locations.

As mentioned above, *MigThread* also supports the checkpointing of individual threads. This is done by invoking the application on the target/restore machine and indicating that a particular thread should be restored. The thread specified through a configuration file is then restored independent of any additional threads. This can be particularly useful in load balancing applications.

A variation on the C^3 system that is capable of checkpointing shared memory systems is dedescribed in [1]. In particular, this system is designed to operate using the OpenMP multi-threading compiler directives [28].

In the shared memory implementation of C^3, variables are saved similarly to the technique described in [18]. Most importantly, the memory pool is still used in order to restore pointer variables to their original locations. This has important implications, as the authors note. In particular, the C^3 checkpoints are generally not portable to heterogeneous architectures due to the fact that the C^3 system performs no data conversion, not even converting between little and big endian.

The most interesting features of C^3 system are its synchronization constructs. The problem is that calls to checkpoint a thread are hints and may not occur each time a thread comes across a checkpoint. Given that no assumptions can be made as to the rate at which threads progress it is possible for two or more threads to deadlock at a checkpoint. For example, consider a two-threaded application where the first thread reaches an application barrier while the second reaches a checkpoint barrier. In such a scenario, neither thread will progress as the first is waiting for the second at the application barrier, and the second is waiting for the first at a checkpoint barrier.

The solution to this problem, according to [1] is to ensure that a checkpoint never crosses an application barrier. In order to do this, C^3 intercepts calls to the OpenMP API and introduces a global **checkpointFlag** variable. When a thread wishes to take a checkpoint it first sets the **checkpointFlag** variable to true and then proceeds to the first checkpoint barrier. Threads that are waiting at an application barrier poll the **checkpointFlag** variable. If they find that it is set to true, the threads waiting at the application barrier immediately begin a checkpoint before returning to the application barrier.

Locks are another problem that must be addressed. The problem is that one thread may hold a mutex and wish to checkpoint, while a second thread may wish to first aquire the mutex before checkpointing. In order for the second thread to perform the checkpoint, the first thread must release its mutex. A proper checkpoint library should ensure that when the application is restored the mutex will be returned to the first thread. To ensure that this occurs, threads are charged with maintaining a table of their own mutexes. Upon restore, each thread checks its table and releases any mutexes that it should not hold.

2.4 Grid/Cluster Application-Level Checkpointing

We now turn to the final type of application-level checkpointing, in particular checkpointing for cluster and grid-based applications. There are a variety of techniques that have been used in order to perform checkpointing in distributed

systems, most of which focuses on message-passing systems. In particular, [29] presents a survey of many techniques that are applicable both to user-level and application-level checkpointing. In this section, two typical coordinated checkpointing algorithms will be discussed. But it is worth noting that, according to [29], other techniques also exist including message logging. In [30] an additional technique is used that essentially turns a distributed system into a shared memory system. The advantage of this technique is that techniques used to checkpoint parallel or shared-memory systems can also be adapted. The disadvantage is that most distributed systems do not allow a task to read and write to the remote address space of another task.

The first checkpointing protocol for message passing systems that will be examined again utilizes the C^3 system. This time, however, the focus is not on the C^3 system itself, but rather the protocol used to guarantee an accurate and consistent global state for MPI programs [26]. A coordinated checkpointing protocol is one in which processes checkpoint together, typically by utilizing a initiator that is responsible for coordinating the protocol. Non-coordinated checkpointing protocols are also possible, some of which are described in [29]. The main problem with the non-coordinated technique is the possibility of their experiencing the "domino effect" due to inconsistent global states. The "domino effect" can lead to the failure of every checkpoint, eventually requiring that the application be restarted [31].

The coordinated technique used in C^3 uses an initiator that is responsible for initiating and coordinating the checkpoint protocol. In C^3 the initiator is simply process 0. When the application is instrumented with the C^3 system more checkpoints are inserted than may be needed. It is therefore the initiator's responsibility to begin the checkpoint when necessary.

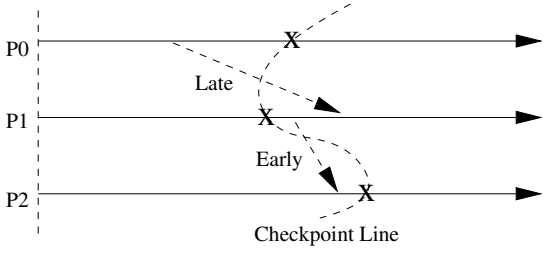

Fig. 2. Illustration of late and early messages

The primary problem with checkpointing message-passing systems is in dealing with both late and early messages. This is a problem in application-level schemes (particularly MPI due to the "tag" that is placed on a message) as there is no guarantee of FIFO message delivery. A late message is one that crosses a checkpoint line in the forward direction. That is, a process sends a message, a checkpoint is taken, and the destination process receives the message after the checkpoint. The problem in this case is that upon resuming from the

intermediate checkpoint, the first process will have already sent the message before the checkpoint and will not resend it after the checkpoint. But the receiving process, which did not receive the message until after the checkpoint, will not receive the message on restart as the sending process will not have re-sent it. Therefore, the global state will be inconsistent.

A similar problem occurs if a message crosses a checkpoint line in the reverse direction. In this case, however, a process will have taken a checkpoint and proceeded to send a message to another process which has not yet taken the checkpoint. Upon restarting from the checkpoint, the receiving process will have already received the message, while the sending process will again re-send the message, resulting in duplicate messages being sent. In order to maintain a consistant global state, such duplicate messages should be suppressed.

Both late and early messages are illustrated in figure 2. An arrow indicates a sent message, where the source is the arrow's origin with the arrow pointing at the destination.

In [26] a 4-phase protocol for C^3 is described that solves the problem of late and early messages. The first phase begins the protocol when the initiator sends a **pleasecheckpoint** message to all processes indicating that a checkpoint should take place when the process reaches its next checkpoint location. Processes at this state are still free to send and receive messages freely.

Phase two begins once a process reaches a checkpoint location. It saves its state including all early messages and begins waiting for, and recording, late messages. When all messages have been received (this is done by comparing the messages sent/received from each process similar to that described in [5]), the process sends a **readytostoprecording** message to the initiator, but still continues to record.

When the initiator receives the **readytostoprecording** from all processes it responds with a **stopRecording** message directed to all processes (phase 3). Phase 4 terminates the protocol when the application stops recording to its checkpoint file. A process stops recording when either it receives the **stopRecording** message from the initiator or when it receives a message from a process that has already stopped recording (since there is no guarantee of FIFO message delivery, etcetera). At this point the process then sends a **stoppedRecording** message back to the initiator. Once the initiator receives the **stoppedRecording** message from all processes the checkpoint can be written to disk and the protocol terminated [26].

In XCAT3 [21] a slightly different protocol is used to checkpoint grid-based applications. Unlike the non-blocking technique used in [26], Krishnan and Gannon describe a blocking approach to grid-based checkpointing. The primary difference between blocking and non-blocking coordinated checkpointing is that in the blocking case a consistant global state is achieved by emptying the communication channel before checkpointing. A high-level description of such a protocol consists of three phases.

1. Empty communications channel.
2. Individual processes checkpoint their local state.
3. Computation resumes.

The checkpointing protocol requires changes to the *XCAT3* architecture, particularly in implementing the application coordinator. The application coordinator essentially serves the same function as the initiator but also provides some additional grid-specific services. In particular, the coordinator provides references into the grid file storage service. This is particularly important in a grid-scenario as a global filesystem such as NFS is not necessarily available, and a checkpoint saved only to local storage would likely prove useless. Therefore a more resilient storage mechanism is needed. In the *XCAT3* system this is provided by a "master storage service" [21].

In the *XCAT3* system a checkpoint is initiated by a user who instructs the application coordinator to being a checkpoint. Upon receiving this, the application coordinator sends a **freezeComponent** message to each mobile component ID. One can think of the mobile component ID as a remote process. When the mobile component receives the **freezeComponent** message it waits for all "remote invocations" to complete.

Once the application coordinator receives a **componentFrozen** message from each component the coordinator can be certain that all communication has been terminated. The coordinator then sends each remote component a reference to an individual storage service on the master storage service as part of a **storeComponentState** message.

Upon receiving the **storeComponentState** message each remote component stores its complete computation state at the location given by the individual storage service. After storing the individual checkpoint at the individual storage service, a storageID is generated and returned to the component. The components then return the storageID to the application coordinator.

The application coordinator stores references to the storageID into its own stable storage so that the checkpoints can be easily located. Upon committing the references to stable storage, the application coordinator sends a **unfreezeComponent** message to each component and computation resumes.

3 Conclusions

Given the current interest in grid-based systems and the potential (indeed likelihood) for a grid to be composed of an array of different architectures and operating systems, it seems likely that the focus of checkpointing researchers will be aimed more towards distributed and grid-based checkpointing than shared-memory or single process checkpointing. With that in mind, it is the opinion of these authors that application-level checkpointing will be the focus of checkpointing for the near futur due primarily to its support for heterogeneity.

References

1. Bronevetsky, G., Marques, D., Pingali, K., Szwed, P., Schulz, M.: Application-level checkpointing for shared memory programs. In: ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, ACM Press (2004) 235–247

2. Milojicic, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. **32**(3) (2000) 241–299
3. Sorin, D.J., Martin, M.M.K., Hill, M.D., Wood, D.A.: Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture, IEEE Computer Society (2002) 123–134
4. Duell, J.: The design and implementation of berkeley lab's linux checkpoint/restart (2003) http://old-www.nersc.gov/research/FTG/ checkpoint/reports.html.
5. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In: Proceedings, LACSI Symposium, Sante Fe, New Mexico, USA (2003)
6. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-transparent checkpoint/restart for mpi programs over infiniband. In: ICPP'06: Proceedings of the 35th International Conference on Parallel Processing, Columbus, OH (2006)
7. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under unix. Technical Report UT-CS-94-242 (1994)
8. Bozyigit, M., Wasiq, M.: User-level process checkpoint and restore for migration. SIGOPS Oper. Syst. Rev. **35**(2) (2001) 86–96
9. Dimitrov, B., Rego, V.: Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. IEEE Transactions on Parallel and Distributed Systems **9**(5) (1998) 459
10. Mascarenhas, E., Rego, V.: Ariadne: Architecture of a portable threads system supporting thread migration. Software- Practice and Experience **26**(3) (1996) 327–356
11. Itzkovitz, A., Schuster, A., Wolfovich, L.: Thread migration and its applications in distributed shared memory systems. Technical Report LPCR9603, Technion, Isreal (1996)
12. Jiang, H., Chaudhary, V.: Process/thread migration and checkpointing in heterogeneous distributed systems. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences. (2004) 282
13. Karablieh, F., Bazzi, R.A.: Heterogeneous checkpointing for multithreaded applications. In: Proceedings. 21st IEEE Symposium on Reliable Distributed Systems. (2002) 140
14. Jiang, H., Chaudhary, V., Walters, J.P.: Data conversion for process/thread migration and checkpointing. In: Proceedings. 2003 International Conference on Parallel Processing. (2003) 473
15. Beguelin, A., Seligman, E., Stephan, P.: Application level fault tolerance in heterogeneous networks of workstations. J. Parallel Distrib. Comput. **43**(2) (1997) 147–155
16. Jiang, H., Chaudhary, V.: On improving thread migration: Safety and performance. In Sahni, S., Prasanna, V.K., Shukla, U., eds.: Proceedings 9th International Conference on High Performance Computing HiPC2002. Volume 2552 of Lecture Notes in Computer Science., Berlin, Germany, Springer-Verlag (2002) 474–484
17. Karablieh, F., Bazzi, R.A., Hicks, M.: Compiler-assisted heterogeneous checkpointing. In: Proceedings. 20th IEEE Symposium on Reliable Distributed Systems. (2001) 56
18. Szwed, P.K., Marques, D., Buels, R.M., McKee, S.A., Schulz, M.: Simsnap: fast-forwarding via native execution and application-level checkpointing. In: INTERACT-8 2004. Eighth Workshop on Interaction between Compilers and Computer Architectures. (2004) 65

19. Strumpen, V.: Compiler technology for portable checkpoints (1998)
20. Lyon, B.: Sun external data representation specification. Technical report, SUN Microsystems, Inc., Mountain View (1984)
21. Krishnan, S., Gannon, D.: Checkpoint and restart for distributed components in xcat3. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing. (2004) 281
22. Ramkumar, B., Strumpen, V.: Portable checkpointing for heterogeneous architectures. In: Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society (1997) 58–67
23. Zhou, H., Geist, A.: "Receiver makes right" data conversion in PVM. In: Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications, IEEE Computer Society (1995) 458–464
24. Zhong, H., Nieh, J.: The ergonomics of software porting: Automatically configuring software to the runtime environment (2006) http://www.cwi.nl/ftp/steven/enquire/ enquire.html.
25. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Collective operations in application-level fault-tolerant mpi. In: ICS '03: Proceedings of the 17th annual international conference on Supercomputing, ACM Press (2003) 234–243
26. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Automated application-level checkpointing of mpi programs. In: PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM Press (2003) 84–94
27. Jiang, H., Chaudhary, V.: Compile/run-time support for thread migration. In: Proceedings International Parallel and Distributed Processing Symposium, IPDPS, IEEE Computer Society (2002) 58–66
28. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. In: IEEE Computational Science and Engineering, IEEE Computer Society (1998) 46–55
29. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3) (2002) 375–408
30. de Camargo, R.Y., Goldchleger, A., Kon, F., Goldman, A.: Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In: Proceedings of the 2nd workshop on Middleware for grid computing, ACM Press (2004) 35–40
31. Agbaria, A., Freund, A., Friedman, R.: Evaluating distributed checkpointing protocols. In: Proceedings. 23rd International Conference on Distributed Computing Systems. (2003) 266