A Heuristic Checkpoint Placement Algorithm for Adaptive Application-Level Checkpointing

Yanqing Ji
Department of Electrical and Computer Engineering
Gonzaga University, Spokane
WA 99258 USA

Hai Jiang
Department of Computer Science
Arkansas State University
Jonesboro, AR 72467 USA

Vipin Chaudhary
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Buffalo, NY 14260 USA

Abstract

Checkpoint/rollback is an effective scheme for fault tolerance and has been widely used to reduce the overall execution time of long-running applications in case of faults. The locations of checkpoints in application programs are critical since the distance between two consecutive ones determines the checkpointing scheme's sensitivity and overheads. If they are too far apart, applications might be insensitive to job failure. That is, the lost computational time between the point of failure and the end of previous checkpoint would be very large. But if they are too close, the related checkpointing overheads will slow down the normal computation. This paper proposes a heuristic checkpoint placement algorithm to improve the checkpointing schemes' performance in terms of sensitivity and flexibility. This heuristic algorithm enables automatic and transparent insertion of checkpoints in user's source code. Experiments on benchmark programs and real applications demonstrate this algorithm's efficiency and sufficiency.

Keywords: Job failure, application-level checkpointing, checkpoint placement algorithm.

1. Introduction

Checkpointing is a common technique for reducing the worst-case execution time of programs in case of faults. It supports fault tolerance by saving computation states to secondary storage and retrieving them later to resume execution after a machine fails or crashes (Gupta, Naik, & Beckman, 2011; Jiang & Chaudhary, 2004; Oliner, Rudolph, & Sahoo, 2006a). Therefore, checkpointing reduces total worst-case execution time of a program by minimizing the lost processing time (Awasthi, Misra, & Joshi, 2010; Daly, 2003). It is essential for high performance computing or long-running applications that mail fail. However, to support checkpointing and minimize overheads, one needs to determine where to insert checkpoints and how to stop and resume computations. A checkpoint is a location in a program where a thread/process can be checkpointed correctly. Since the overheads associated with constructing, saving, and retrieving computation states are not negligible (Greg Bronevetsky, Daniel Marques, Keshav Pingali, Radu Rugina, & McKee, 2008; Jiang, Chaudhary, & Walters, 2003; John Paul Walters & Chaudhary, 2009), finding appropriate checkpoints is essential.

The distance between two consecutive checkpoints determines the checkpointing algorithm's sensitivity and overheads. If they are too far apart, the applications might be too insensitive to job failure since the lost processing time could be very long. But if they are too close, the related overheads will slow down the actual computation (Ziv & Bruck, 1997). The trade-off between the re-processing time and checkpointing overheads leads to an optimal checkpoint placement. Several attempts have been made to find the optimal checkpoint intervals under certain computing environments and assumptions (Daly, 2003; Oliner, Rudolph, & Sahoo, 2006b; Young, 1974). However, most such research focused on kernel or user-level checkpointing, i.e., memory execution image can be treated as the computation state and dumped by certain system or library calls. With such assumption, checkpointing can take place anywhere anytime with invariant costs. Thus, proper checkpoint interval can be calculated to provide required fault tolerance functionality without incurring much state saving/recovering overhead.

However, in heterogeneous environments, checkpointing becomes much more complicated. First, there is no universal system call or library call to dump computation states since different operating systems and architectures use different formats (Manivannan, 2008). There is no sign for a possible convergence. Possibly, we will see an even wider diversity in the future. For portability, computation states have to be constructed at higher level, such as application level. Some application-level checkpointing packages such as *MigThread* abstract computation states in applications for portability (Jiang & Chaudhary, 2004). When checkpointing happens, the application can calculate its own state without any help from the system and kernel. MigThread can augment programs by inserting state construction/retrieval statements during pre-compilation. The users then need to re-compile this augmented code. One major restriction of application-level checkpointing is that the computation states cannot be constructed/retrieved anywhere, anytime. Checkpointing can occur only at predefined locations inserted in the source code and determined at compile time. Thus, it is harder to determine the checkpoint interval/period (which is a runtime issue) in application-level checkpointing than kernel or user level checkpointing. Therefore, both timing and location issues should be considered at both compile time and runtime. For heterogeneous computing such as Grids, application-level approach is the only option (Babar Nazir, Kalim Qureshi, & Manuel, 2008; Maria Chtepen et al., 2009).

In this paper, we propose a heuristic checkpoint placement algorithm for applications whose computation state can be constructed at the application level. Our approach is to aggressively insert a lot of *potential* checkpoints into users' programs. Whether a *potential* checkpoint will be actually activated (i.e. checkpointing will be triggered) is decided by a scheduler (or server) which could utilize any optimal checkpoint interval estimation method (e.g Young's law (Young, 1974)) to determine the actual checkpoint intervals. For example, a timer can be set on a server to remind applications that it is time for checkpointing. Our algorithm describes how to insert these *potential* checkpoints and tries to make programs sensitive enough. This algorithm can transparently insert checkpoints into the source code so that, at run time, the computation can be properly checkpointed. We make the following specific contributions in this paper:

- Identify checkpoint placement issues in checkpointing.
- Propose a heuristic scheme to transparently insert checkpoints.
- Evaluate effectiveness and performance of this scheme.

The remainder of this paper is organized as follows: Section 2 provides an overview of application-level thread/process checkpointing. Section 3 identifies the issues involved in checkpoint placement. We describe the details of our algorithm in Section 4. In Section 5, we give experimental results on benchmark programs and real applications. Section 6 is an overview of related work. We wrap up with conclusions and future work in Section 7.

2. Thread/Process Checkpointing

In this section we briefly describe an application-level checkpointing scheme MigThread that is already suited for this work. We explain some of the reasons of this choice in Section 4.

2.1 MigThread

Since *MigThread* provides application-level migration and checkpointing functionalities for sequential and parallel computations in heterogeneous or Grid computing environments (Jiang & Chaudhary, 2004), it is selected for our heuristic algorithm design and experiments. In *MigThread*, both coarse-grained processes and fine-grained threads are supported and both migration and checkpointing are available. But in this paper we only utilize its checkpointing functionality. For a certain process, its threads can be simultaneously checkpointed to secondary storage. For process checkpointing, all internal threads as well as their shared global data are processed together.

Typically, computation states consist of process data segments, stacks, heaps and register contents. In *MigThread*, the computation state is moved out from its original location (libraries or kernels) and abstracted up to the language level. Thus, the physical state is transformed into a logical form to achieve platform-independence. Both the portability and the scalability of stacks are improved.

MigThread consists of two parts: a preprocessor and a run-time support module. The preprocessor is designed to transform user's source code into a format from which the run-time support module can construct computation states precisely and efficiently.

A powerful preprocessor can improve the transparency of application-level checkpointing. The run-time support module constructs, saves, and restores computation states dynamically as well as provides other run-time safety checks (Jiang & Chaudhary, 2004).

2.2 Data Conversion Scheme

The major obstacle preventing application-level checkpointing from achieving widespread use is the complexity of adding transparent checkpointing to systems originally designed to run stand-alone. Heterogeneity further complicates this situation. To support heterogeneity, computation states constructed on one platform need to be interpreted by another. In *MigThread*, a data conversion scheme, called Coarse-Grain Tagged "Receiver Makes Right" (CGT-RMR) (Jiang, et al., 2003), is used to tackle data alignment and padding physically, convert data structures as a whole, and eventually generate a lighter workload compared to existing standards. It accepts ASCII character sets, handles byte ordering, and adopts IEEE 754 floating-point standard because of its dominance in the market.

2.3 Checkpointing Safety

Checkpointing safety concerns the correctness of the resumed computation. In other words, computation states should be constructed precisely, and restored correctly on similar or different machines. The major identified unsafe factors come from unsafe type systems (such as the one in C programming language), incompatible data conversion, and third-party library calls. In *MigThread*, a pointer inference algorithm is proposed to detect the hidden pointers caused by the unsafe type system. The data conversion scheme, CGT-RMR, supports aggressive data conversion and aborts state restoration only when "precision loss" event occurs.

3. Issues Affecting Checkpoint Placement

There are two important issues that affect the choice of checkpoint placement methods. First, checkpoint placement depends on the programming model used in the applications. For example, in some multithreaded or distributed computing applications with synchronization operations, checkpointing should only occur at these synchronization points, such as barriers in software Distributed Shared Memory systems (DSMs). Otherwise the data could be inconsistent. Second, since we are trying to find a general checkpoint placement scheme at the *application-level*, the underlying hardware and compilers could dramatically affect whether a heuristic method or a quantitative approach is chosen. We discuss these issues in the following subsections.

3.1 Programming Paradigms

For sequential applications, checkpoints can be inserted almost anywhere in the program. Even though we mainly focus on the checkpointing in sequential applications in this paper, our proposed algorithm can be extended to parallel programming paradigms. In parallel and distributed computing environments, the basic strategy still works. However, additional complexity due to consistency issues need to be handled. *MigThread* is a package that can take care of the consistency of computation states. The shared-address-space programming paradigm (such as multithreading) is popular because of its simplicity. However, when using this model in distributed systems, memory consistency models need to be applied to keep consistent data copies across multiple processors. Since traditional parallel applications adopt the sequential memory consistency model, the data is always in a consistent state. Thus, no restriction is placed on the locations where the checkpoints are inserted. In some advanced parallel computing environments, e.g. modern Distributed Shared Memory (DSM) systems (Roy & Chaudhary, 1998; Taesoon Park & Yeom, 2000; Zhou et al., 1997), relaxed memory models are used to reduce both the number of messages and the amount of data transferred between processors for better performance.

Under such aggressive models, some virtually shared data could be in inconsistent states when they are between two synchronization points (barriers), i.e., their copies on physically different machines might have different values. If checkpointing takes place at these machines at any non-synchronization points, and inconsistent local copies of data are accessed (especially read) later, the resumed computation could be incorrect. To ensure correctness, checkpointing can be allowed only at synchronization points or barriers. To improve sensitivity, the preprocessor could insert light/pseudo-barriers for synchronizing the progress of all threads. When checkpointing is required, real barriers are invoked to synchronize both progress and data. Such light/pseudo-barriers could incur extra overheads in some aggressive parallel computations; however, they will improve adaptability by adding more possible checkpointing locations. Message passing (such as library MPI) is another commonly used programming paradigm.

In this model, if migration/checkpointing occurs between *send* and *receive* operations, the transit data may not be accounted and thus lead to data inconsistency. Therefore, in the current version of *MigThread*, migration/checkpointing can only be allowed at consistent points (before the *send* operation and after the *receive* operation). Our checkpoint placement algorithm can handle different parallel/distributed programming paradigms (e.g DSM, MPI), but to make our description clear and for ease of implementation, we focus on sequential applications in this paper.

3.2 Quantitative vs. Heuristic Methods

We have tried more quantitative methods that statically estimate the execution time of the user program and then insert checkpoints using existing optimal checkpoint interval estimation methods. In real-time or embedded system where the missing of a deadline will result in a catastrophic failure, a lot of work has been done regarding statistical estimation of the execution time (Brandolese, Fornaciari, Salice, & Sciuto, 2001; Giusto, Martin, & Harcourt, 2001; Malik, Martonosi, & Li, 1997). Most investigations focus on WCET (Worst Case Execution Time) since it is hard to know the execution path at compile time.

One of the major difficulties in estimating execution time is to find loop bounds which are usually determined by the input of programs. For most current solutions, users are required to input these loop bounds (Li, Stewart, & Fuchs, 1994). This method is viable since programs in real-time or embedded systems are usually very simple and small. But for complex applications in scientific computing, it is impossible to ask users to input the upper bound for each loop. Another drawback of these methods is that the algorithms are machine-dependent. Since we are targeting an application-level and machine-independent checkpointing algorithm, such solutions are not applicable. Furthermore, caches and compiler optimizations, such as loop unrolling and software pipelining, make it harder to estimate the execution time of a program. Our experience shows that a quantitative approach is unsuitable for achieving stable results at application-level.

4. Heuristic Algorithm for Checkpointing

Since *MigThread* is implemented at application-level, we have to insert certain code into user programs in order to enable checkpointing functionality. The challenge is to detect proper locations in the code and treat them as checkpoints while maintaining an appropriate distance between two checkpoints.

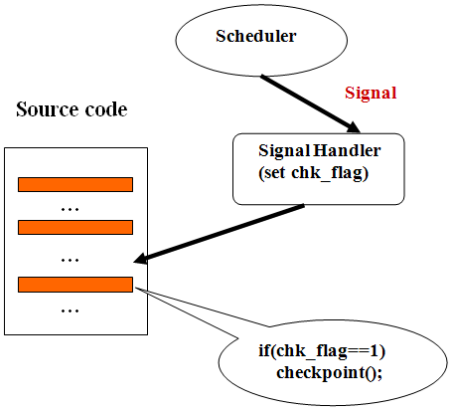

Figure 1. General mechanism for checkpoint placement

Instead of selecting actual checkpoints, our solution is to aggressively insert *potential checkpoints*. Whether these checkpoints will be actually executed is determined by a scheduler (or server) which is executed in the background, as shown in Figure 1. That is, at compile time, the preprocessor inserts a lot of *potential checkpoints*. At run time, once the scheduler determines that a thread/process needs to be checkpointed according to any optimal checkpoint interval algorithm, it sends a signal to *MigThread*. The signal handler will set the checkpoint flag (*chk_flag*), and the corresponding thread/process will be actually checkpointed at the next *potential checkpoint*. We will give more details in the next two subsections. Another feature of our algorithm is that the computation state is constructed only when checkpointing indeed happens. The preprocessor of *MigThread* has collected all related variables at the beginning of functions whereas other systems (Chanchio & Sun, 2001; Ramkumar & Strumpen, 1997; Ssu & Fuchs, 1998) need to report variables one-by-one at checkpoints during migration/checkpointing. Therefore, the cost of *MigThread* is much lower so that more checkpoints can be inserted. And this is one of the main reasons to select *MigThread* system for our algorithm here.

4.1 Potential Checkpoints

The code that establishes a checkpoint is a subroutine named *checkpoint()*. The preprocessor of *MigThread* has to be improved so that it can insert checkpoints into user's source code. Detecting proper locations for checkpoints at compile time might not always be possible since some code relies on dynamic inputs. In many applications, loops consume most of the execution time in a program. If their bounds are not known statically, it is hard to insert checkpoints properly. For example, in Figure 2, the preprocessor may not be able to determine where the *checkpoint()* subroutine should be inserted.

```
for (i = 0; i < upperBound; i++) {
    sum += func(i);}
printf("%d", sum);
```

Figure 2. Loops with unknown upper bound

If the preprocessor inserts *checkpoint()* subroutine inside the loop, the thread/process checkpointing might occur during each iteration. With a simple iteration code in this example, checkpointing can take place too frequently. Since the overhead introduced by checkpointing activity is not negligible, frequent checkpointing will greatly affect the performance of user's application. However, if the preprocessor places the *checkpoint()* call after the loop, the interval between two checkpoints could be so long as to make the system insensitive to failures. We solve this dilemma by inserting a *potential checkpoint* in each loop, whereas the scheduler determines whether a checkpoint needs to be actually set (Figure 1). In this way, the code block in Figure 2 can be transformed as in Figure 3. If no checkpointing happens, the only operation is to check a flag variable in the runtime support module linked with applications.

```
for (i = 0; i < upperBound; i++) {
    sum += func(i);
    if (chk_flag == 1) {
        checkpoint();
    }
 }
printf("%d", sum);
```

Figure 3. Potential checkpoints in loops with unknown upper bound

Note that inserting *if* statements, especially in a loop, may degrade the performance of user's program on modern microprocessors since it affects compiler optimizations. However, this issue is unavoidable for application-level solutions. And our experiments in Section 5 show that the overhead is acceptable.

4.2 Potential Checkpoint Placement Rules

To enable checkpointing, we modified the preprocessor of *MigThread* to analyze the source code and automatically insert appropriate *potential checkpoints* accordingly. Then, the transformed code will be recompiled and linked with *MigThread* run-time library. Thereafter, the program is ready for checkpointing.

To insert *potential checkpoints*, the preprocessor has to analyze the structure and different components of user's source code. Usually, programs consist of loops, common non-loop code blocks, function calls, and library calls. Besides, the preprocessor has to take care of some special statements such as return, exit, and so on. We will describe the general rules for inserting *potential checkpoints* in the following subsections.

4.2.1 Loops

Based on the analysis in Section 3, we have to specially take care of loops in user's source code. In general, we apply the following rules to insert the *potential checkpoints*.

Rules for loops:

- One *potential checkpoint* is inserted right after the last statement of every loop. This rule guarantees the sensitivity of our system since it avoids long interval between two consecutive *potential checkpoints*.
- In case of nested loop, *potential checkpoints* are inserted inside the innermost loop since the code in inner loop will also be executed in its outer loop.
- Within the same outer loop, if there are multiple nested loops in parallel, one *potential checkpoint* needs to be inserted in each of them.

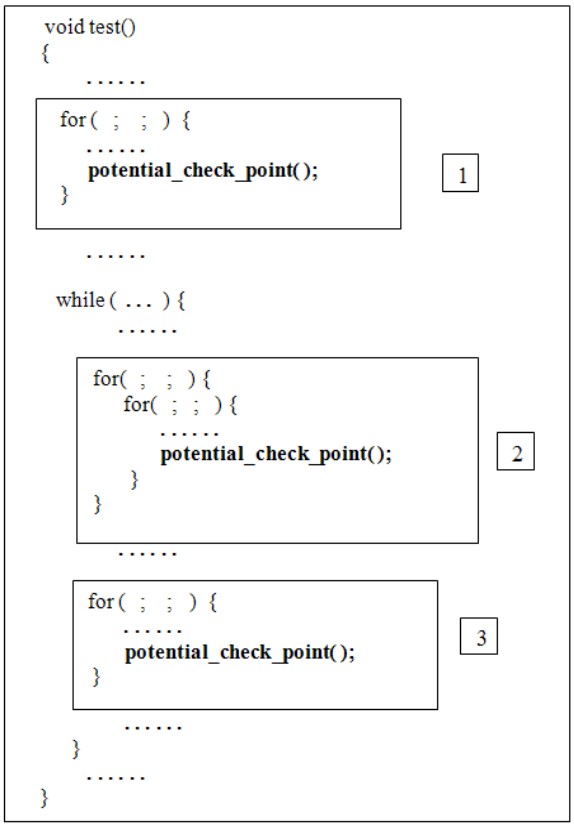

Figure 4. Potential checkpoints for loops

Figure 4 shows our scheme for inserting *potential checkpoints* into loops. Here we use potential_check_point() to represent the code of a *potential checkpoint* (i.e. the *if* block shown in Figure 1 and 3). For loop 1, we insert one *potential checkpoint* at the last line of this loop. Loop 2 is a nested loop, thus the *potential checkpoint* is inserted inside the innermost loop. Loop 3 is in parallel with loop 2 and both of them are inside the same outer loop (the while loop). We also insert a *potential checkpoint* in loop 3. The key idea is that we insert at least one *potential checkpoint* for each loop since we might not know the loop bounds at compile time.

4.2.2 Non-Loop Code and Functions

We argue that the non-loop code is never too long in scientific computations. The experiments in Section 5 also verify this notion. Therefore, we can ignore non-loop code when we try to find the proper place for *potential checkpoints*. However, even if there is no loop in a subroutine, recursive functions and nested calls could consume a long period of time. So, in this case, we also need to insert *potential checkpoints*.

In our scheme, branch instructions are also considered as non-loop code. However, since branch instructions are often used together with “exit” and “return” instructions which may result in different execution path, we have to make sure there is at least one *potential checkpoint* in each path. Otherwise, long intervals could exist through recursive calls or nested calls.

In summary, we apply the following rules to insert *potential checkpoints* in non-loop code and functions.

Rules for non-loop code and functions:

- Non-loop instructions including branch instructions will be ignored since they usually do not consume a lot of time.
- For each subroutine, at least one *potential checkpoint* is inserted. If no loop exists, we insert the *potential checkpoint* at the end of that subroutine.
- To ensure at least one *potential checkpoint* for each execution path, we insert a *potential checkpoint* before any "break" or "return" statement.

Figure 5 gives our scheme for dealing with non-loop code and function calls. In this segment of code, function *subsubroutine()* is called by *subroutine()* which, in turn, is called by another function *test()*. Since there is no loop and "return" in *subroutine()*, we insert one *potential checkpoint* at the end of this function. However, in function *subsubroutine()*, the *potential checkpoint* is added before the "return" instruction. In function *test()*, one *potential checkpoint* is inserted right before the "return" instruction inside the "if" structure because the branch instruction determines a different path. In addition, another *potential checkpoint* is inserted at the end of *test()* function since the first *potential checkpoint* may not be actually executed.

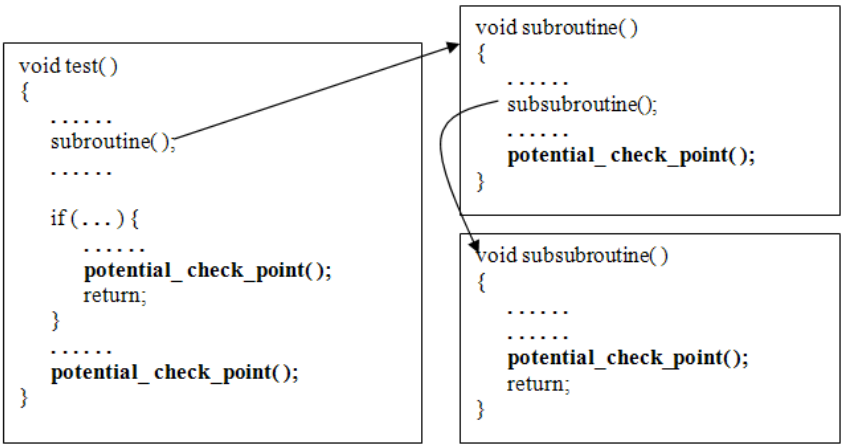

Figure 5. Potential checkpoints for sequential code and function call

4.2.3 Library Call and I/O Operations

Library calls can cause problems for all application-level checkpointing schemes including *MigThread* since all these schemes require access to source code. Without the source code of libraries, the preprocessor cannot insert *potential checkpoints* into the library functions. Actually, this is one of the main disadvantages of application-level checkpointing schemes (Li, et al., 1994; Sun, Naik, & Chanchio, 1999). However, to achieve better portability of the application-level approach, it is reasonable to give up the sensitivity during the third-party library call procedures. Luckily, the execution time of most library calls is relatively short. I/O operations also have the potential of increasing the time between two consecutive *potential checkpoints*. The reason is that the cost of I/O operations depends on the data volume and external factors such as network bandwidth. Our assumption is that I/O is not a proper time for checkpointing.

5. Experimental Results and Performance Analysis

To evaluate our checkpoint placement mechanism, Matrix Multiplication, Molecular Dynamics (MD) simulation and several applications from the SPLASH-2 application suite are chosen for experiments. SPLASH-2 is selected because it is one of the most widely accepted benchmarks. Matrix Multiplication is small, but it is a representative computation-intensive application. Even though we are primarily interested in the overhead ratio and the size of the application does not matter; we still want to know how our scheme affects large real applications. That is why we chose a Molecular Dynamics (MD) application which is used to study friction forces of sliding hydroxylated α-aluminum oxide surfaces (Jin, Song, & Hase, 2000).In this paper, since we are concerned with the overhead introduced by the *potential checkpoints*, we did experiments with sequential applications on a single machine. We are also interested in the sensitivity of our checkpoint placement mechanism.

5.1 Overhead of Potential Checkpoints

Our first experiment is to test the overall overheads associated with our algorithm. In this experiment, the actual checkpointing overheads are ignored by intentionally setting all checkpointing flags (*chk_flag*) to 0. In this case, the overhead tested comprises of two parts: the first part is the time for checking the checkpointing flag at each *potential checkpoint*; the second part is the overhead introduced by inserting *potential checkpoints* since inserting statements into a loop will affect the pipeline and cache usages of the original program.

Table 1 *Potential checkpoints* overhead in real applications

Program	Input size	Exec. time without checkpoints (*us*)	Exec. time with checkpoints (*us*)	Overhead ratio (%)
FFT	1,024 points	3,574	3,620	1.287
LU-c	512 x 512	2,270,464	2,293,040	0.994
LU-n	128 x 128	40,135	40,754	1.542
MatMult	128 x 128	146,810	149,883	2.093
RADIX	262,144 keys	918,865	921,939	0.334
MD	5,286 atoms	18,823,604	18,903,475	0.424

Table 1 shows the overheads of six applications. We are interested in the overhead ratio R, which is defined as the ratio between the average overhead and the execution time of original program. In other words

$$R = (t_1 - t_0)/t_0 \quad (1)$$

where t_1 and t_0 represent the execution time of the application with *potential checkpoints* and without *potential checkpoints*, respectively.

From Table 1, we can see that matrix multiplication is the only application whose overhead ratio is greater than 2% because it is a computation-intensive application with many small loops. For about half of the applications, their overheads are less than 1%. Therefore, the overhead introduced by our algorithm is acceptable. Intuitively, real applications suitable for checkpointing usually execute for a long time, and some of them even run for several weeks or months. In order to test the scalability of the proposed mechanism in terms of the input size of an application, we randomly choose FFT and LU-c from the above six applications for experiments. For each application, the same experiments are performed, but with different input sizes. Their overhead ratios against input sizes are shown in Figure 6.

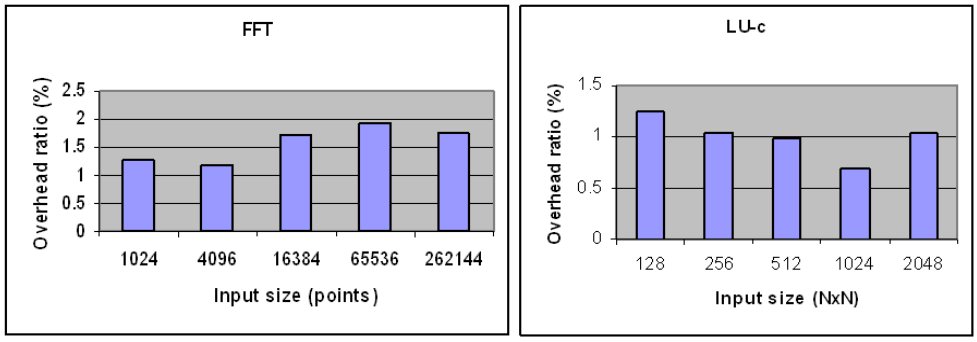

Figure 6. Scalability of the algorithm on two applications

We can see that there is no significant increase of the overhead ratio when the input size increases. Since most *potential checkpoints* are inserted inside loops in a program, the increase of the input size will cause the increase of loop bounds, and thus proportionally increase the overheads associated with the *potential checkpoints*. As a result, the overhead ratio will keep a relative constant value (equation [1]). Therefore, this algorithm is also applicable to large problems.

5.2 Sensitivity Analysis

We are concerned with the scheme's response time, i.e., the time span between the scheduler's checkpointing decision-making and the occurrence of actual checkpointing. The response time t_r can be expressed as follows:

$$t_r = t_s + t_p/2 \quad (2)$$

where t_s is the startup time consumed by sending and receiving a signal and setting the checkpointing flag in the signal handler, and t_p is the average time between two consecutive *potential checkpoints*.

Although checkpointing requests can be generated by the scheduler at any time, in *MigThread*, actual checkpointing can only take place at discrete *potential checkpoints*. The average polling time between two consecutive *potential checkpoints* is $t_p/2$. Thus, the response time, from the moment of checkpointing decision-making to the moment of actual checkpointing, is $t_s + t_p/2$. Since t_s is usually very small, response time t_r mainly depends on t_p.

Table 2. Potential checkpoint distribution

Program	Input size	Potential checkpoint intervals in certain time ranges (*us*)					
		< 10	$10 \sim 10^2$	$10^2 \sim 10^3$	$10^3 \sim 10^4$	$10^4 \sim 10^5$	$> 10^5$
FFT	1,024 points	24,778	15	3	1	0	0
LU-c	512 x 512	48,259,893	4,898	553	148	0	0
LU-n	128 x 128	787,569	96	17	4	0	0
MatMult	128 x 128	2,113,415	227	18	7	0	0
RADIX	262,144 keys	2,368,721	810	30	10	0	0
MD	5,286 atoms	137,443,856	62,050	956	418	0	0

In this experiment, we try to figure out the time distribution of all the intervals between two consecutive *potential checkpoints* in a program. For each application, the number of intervals within different time ranges is recorded, as shown in Table 2. It is clear that more than 99.99% intervals are less than 10 *us*, and no one is greater than 10^4 *us*. Therefore, this experiment is consistent with our previous prediction: the non-loop code is never too long, and within the same outer loop, the non-loop part (other than the inner loops) is not very long.

However, are the intervals ranging between $10^3 \sim 10^4$ *us* acceptable or too long, even though they are very rare? Actually, the acceptable distance between two consecutive *potential checkpoints*, or the frequency of *potential checkpoints* in a program, is governed by the tolerance to the actual checkpointing cost (Sun, et al., 1999). This cost (in terms of time) should not exceed a certain fraction of the time spent on useful work since the last checkpointing. If the response time is less than or comparable to the checkpointing cost, meaning that the response time is much less than the time spent on useful work, it is acceptable.

According to the homogeneous thread migration applications used in (Jiang & Chaudhary, 2002), for most applications the migration overhead is about several *ms*, which is comparable to the worst interval between two consecutive *potential checkpoints* as shown in Table 2. In MigThread, the implementation of checkpointing is similar to that of migration. The only difference is that checkpointing saves computation states into secondary storage instead of migrating them to another machine. Thus, we argue that the overhead of checkpointing should be of the similar order as the overhead of migration. We expect that the cost of process checkpointing is to be bigger than thread checkpointing due to larger stacks and heaps. In a heterogeneous environment, since *MigThread* has to handle the data conversion between different platforms, the cost will only increase. Based on the above facts, the response time, and hence the sensitivity of the proposed checkpoint placement algorithm is acceptable.

In order to test whether the *potential checkpoint* distribution is scalable, same experiments on FFT, LU-c, MatMult and RADIX are conducted as above, but with different input sizes for each application. The experimental results are presented in Tables 3, 4, 5 and 6. Other applications follow the same trend. These tables show that the increase of input size does not generate longer intervals, and if the number of intervals within certain range is big enough, its increase is statistically proportional to the increase of input size (as shown in the left two columns or the last three rows).

Table 3. Scalability of potential checkpoint distribution for FFT

Input size	*Potential checkpoints* intervals in certain time ranges (*us*) – FFT					
	< 10	$10 \sim 10^2$	$10^2 \sim 10^3$	$10^3 \sim 10^4$	$10^4 \sim 10^5$	$> 10^5$
2^{10} points	24,778	15	3	1	0	0
2^{12} points	110,968	61	5	2	0	0
2^{14} points	492,161	236	12	4	0	0
2^{16} points	2,163,463	976	23	10	0	0
2^{18} points	9,436,658	3,975	83	39	0	0
2^{20} points	40,884,903	16,230	329	166	0	0

Table 4. Scalability *of potential checkpoint* distribution for LU-c

Input size	*Potential checkpoints* intervals in certain time ranges (*us*) – LU-c					
	< 10	$10 \sim 10^2$	$10^2 \sim 10^3$	$10^3 \sim 10^4$	$10^4 \sim 10^5$	$> 10^5$
64 x 64	104,053	21	3	1	0	0
128 x 128	787,695	106	11	4	0	0
256 x 256	6,122,137	821	37	25	0	0
512 x 512	48,259,893	4,898	287	148	0	0
1024 x 1024	383,206,760	37,725	27,08	1,417	0	0

Table 5. Scalability of *potential checkpoint* distribution for MatMult

Input size	*Potential checkpoints* intervals in certain time ranges (*us*) – MatMult					
	< 10	$10 \sim 10^2$	$10^2 \sim 10^3$	$10^3 \sim 10^4$	$10^4 \sim 10^5$	$> 10^5$
32 x 32	33,818	6	2	1	0	0
64 x 64	266,252	45	7	3	0	0
128 x 128	2,113,415	227	18	7	0	0
256 x 256	16,841,117	1,719	109	66	0	0
512 x 512	134,465,435	13,473	952	527	0	0

Table 6. Scalability of *potential checkpoint* distribution for RADIX

Input size	*Potential checkpoints* intervals in certain time ranges (*us*) – RADIX					
	< 10	$10 \sim 10^2$	$10^2 \sim 10^3$	$10^3 \sim 10^4$	$10^4 \sim 10^5$	$> 10^5$
4,096 keys	47,112	17	2	1	0	0
16,384 keys	157,666	54	4	2	0	0
65,526 keys	599,883	200	10	4	0	0
262,144 keys	2,368,721	810	30	10	0	0
1,048,576 keys	9,444,073	3,245	94	49	0	0

6. Related Work

Regarding how to insert checkpoints (or migration points), three methods have been proposed. The first approach is that checkpoints are inserted by users or initiated at a barrier (Abdel-Shafi, Speight, & Bennett, 1999). This method is straightforward, but it brings undue burden on inexperienced programmers who do not know the structure and workload of their applications. And for some large and complex applications where many developers are involved, it is very difficult to insert checkpoints by users. SNOW (Sun, et al., 1999) handles migration points by counting the number of floating point operations. That is, it inserts a migration point after a certain number of floating point operations. Since we do not always know the upper bound of a loop at compile time, this scheme has difficulties in counting the operations inside a loop. Furthermore, this scheme is not applicable to non-scientific applications where most operations may not be floating point operations. Our discussion in Section 3.2 has shown the difficulties of a quantitative method because of caches and compiler optimizations. Therefore, such approach might be inaccurate under many circumstances.

The checkpoint placement approach in (Li, et al., 1994) is similar to ours, but it has some limitations. Their approach also inserts *potential checkpoints* inside loops, but instead of using a scheduler, it uses a counter to determine when checkpointing actually occurs. The counter is initially set to a value, called "*reduction factor*", and it is reduced by one for each loop. When the counter is equal to zero, the program does actual checkpointing. This scheme can only insert *potential checkpoints* at certain *sparse* points, and its counter calculation causes larger overhead than checking a variable in our scheme. With many small loops or loops with unknown upper bounds, checkpointing might not take place for a long time. Therefore, this method is insensitive to failures since checkpointing is only allowed at *sparse* points. One of the major advantages of our proposed scheme is that it is generic for both thread/process checkpointing. Since it can tolerate more checkpoints, the applications can be more sensitive to their dynamic situations. On the contrary, other schemes only allow checkpointing at very *sparse* points. Furthermore, our scheme has the potential to work with more complex schedulers, which could be very important for meta-computing or Grid computing (Cicerre, Madeira, & Buzato:, 2006; Zhu, Xiao, Xu, & Ni, 2006). Grids are typical heterogeneous computing environments. Portability is a critical issue.

Thus, application-level checkpointing is the only option since on different platforms computation states will be represented differently. Common kernel-level and user-level approaches are only applicable on homogeneous clusters. Our target is application-level approach which will be adopted by Grids because of its heterogeneity nature.

7. Conclusions and Future Work

We have proposed a heuristic checkpoint placement algorithm which can be incorporated into application-level checkpointing packages, such as *MigThread.* According to our experiments on SPLASH-2 benchmark programs and the Molecule Dynamics simulation application, our algorithm inserts sufficient *potential checkpoints* for high sensitivity. However, some programmers might apply unstructured programming styles which cause sparse *potential checkpoints*. For such applications, users can always insert *potential checkpoints* manually. At each *potential checkpoint*, the scheduler will decide whether an actual checkpointing should occur or not according to system workload, application requirements, and optimal checkpoint interval algorithms. Contrary to the approaches adopted in many other research projects, our algorithm is more flexible, realistic, adaptive and sensitive to system requirements. We have demonstrated the efficacy of our algorithm using several benchmarks and real applications. We are currently investigating a more elaborate scheduler to make a choice of data, thread or process migration/checkpointing for better data locality and communication minimization. Such scheduling processes will further indicate if the *potential* check*points* inserted by this algorithm are sufficient. This algorithm helps improve applications' adaptability and sensitivity in heterogeneous distributed computing environments.

Acknowledgment

This research was supported in part by NSF grants IGERT 9987598, NSF MRI 9977815, and NSF ITR 0081696.

References

Abdel-Shafi, H., Speight, E., & Bennett, J. K. (1999). *Efficient user-level thread migration and checkpointing on Windows NT clusters.* Paper presented at the Proceedings of the 3rd USENIX Windows NT Research Symposium, Seattle, Washington

Awasthi, L. K., Misra, M., & Joshi, R. C. (2010). A weighted checkpointing protocol for mobile distributed systems. *Int. J. Ad Hoc Ubiquitous Comput., 5*(3), 137-149.

Babar Nazir, Kalim Qureshi, & Manuel, P. (2008). Adaptive checkpointing strategy to tolerate faults in economy based grid. *The Journal of Supercomputing, 50*(1), 1-18.

Brandolese, C., Fornaciari, W., Salice, F., & Sciuto, D. (2001). *Source-Level Execution Time Estimation of C Programs.* Paper presented at the IEEE International Workshop on Hardware Software Co-Design, Copenhagen, Denmark.

Chanchio, K., & Sun, X.-H. (2001, April). *Data Collection and Restoration for Heterogeneous Process Migration.* Paper presented at the Proceedings of 21st International Conference on Distributed Computing Systems, San Francisco, CA.

Cicerre, F. R. L., Madeira, E. R. M., & Buzato:, L. E. (2006). Structured process execution middleware for Grid computing. *Concurrency and Computation: Practice & Experience, 18*(6), 581 - 594.

Daly, J. (2003). *A model for predicting the optimum checkpoint interval for restart dumps.* Paper presented at the Proceedings of the International Conference on Computational Science.

Giusto, P., Martin, G., & Harcourt, E. (2001). *Reliable Estimation of Execution Time of Embedded Software.* Paper presented at the Proceedings of the Conference on Design, Automation, and Test in Europe, Munich, Germany.

Greg Bronevetsky, Daniel Marques, Keshav Pingali, Radu Rugina, & McKee, S. A. (2008). *Compiler-Enhanced Incremental Checkpointing for OpenMP Applications*. Paper presented at the Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming.

Gupta, R., Naik, H., & Beckman, P. (2011). Understanding Checkpointing Overheads on Massive-Scale Systems: Analysis of the IBM Blue Gene/P System. *Int. J. High Perform. Comput. Appl., 25*(2), 180-192.

Jiang, H., & Chaudhary, V. (2002, August). *MigThread: Thread Migration in DSM Systems.* Paper presented at the Proceedings of the Workshop on Compile/Runtime Techniques for Parallel Computing, held with International Conference on Parallel Processing, Vancouver, Canada.

Jiang, H., & Chaudhary, V. (2004, January 5-8). *Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems.* Paper presented at the Proceedings of the 37th Hawaii International Conference on System Sciences, Hawaii.

Jiang, H., Chaudhary, V., & Walters, J. P. (2003, October 6-9). *Data Conversion for Process/Thread Migration and Checkpointing.* Paper presented at the Proceedings of International Conference on Parallel Processing Kaohsiung, Taiwan.

Jin, R. Y., Song, K., & Hase, W. L. (2000). Molecular Dynamics Simulations of the Structures of Alkane/Hydroxylated α-Al2O3(0001) Interfaces. *Journal of Physical Chemistry B, 104*(12), 2692–2701.

John Paul Walters, & Chaudhary, V. (2009). Replication-Based Fault Tolerance for MPI Applications *IEEE Transactions on Parallel and Distributed Systems, 20*(7), 997-1010.

Li, C.-C. J., Stewart, E. M., & Fuchs, W. K. (1994). Compiler assisted full checkpointing. *Software - Practice and Experience, 24*(10), 871-886.

Malik, S., Martonosi, M., & Li, Y.-T. S. (1997, June). *Static timing analysis of embedded software.* Paper presented at the Design Automation Conference, Anaheim, CA.

Manivannan, D. (2008). *Checkpointing and rollback recovery in distributed systems: existing solutions, open issues and proposed solutions*. Paper presented at the Proceedings of the 12th WSEAS international conference on Systems.

Maria Chtepen, Filip H.A. Claeys, Bart Dhoedt, Filip De Turck, Piet Demeester, & Vanrolleghem, P. A. (2009). Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids. *IEEE Transactions on Parallel and Distributed Systems, 10*(2), 181-192.

Oliner, A. J., Rudolph, L., & Sahoo, R. K. (2006a). *Cooperative checkpointing: a robust approach to large-scale systems reliability*. Paper presented at the Proceedings of the 20th annual international conference on Supercomputing.

Oliner, A. J., Rudolph, L., & Sahoo, R. K. (2006b, June 28). *Cooperative checkpointing: a robust approach to large-scale systems reliability.* Paper presented at the Proceedings of the 20th Annual International Conference on Supercomputing, Cairns, Australia.

Ramkumar, B., & Strumpen, V. (1997, 24-27 Jun). *Portable Checkpointing for Heterogeneous Architectures.* Paper presented at the 27th International Symposium on Fault-Tolerant Computing, Seattle, WA.

Roy, S., & Chaudhary, V. (1998). *Strings: A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters*. Paper presented at the Proc. of IEEE Conf. on High Performance Distributed Computing.

Ssu, K.-F., & Fuchs, W. K. (1998, 23-25 Jun). *PREACHES - Portable Recovery and Checkpointing in Heterogeneous Systems.* Paper presented at the Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing Munich, Germany.

Sun, X.-H., Naik, V. K., & Chanchio, K. (1999, March 22-24). *A Coordinated Approach for Process Migration in Heterogeneous Environments.* Paper presented at the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX.

Taesoon Park, & Yeom, H. Y. (2000). A Low Overhead Logging Scheme for Fast Recovery in Distributed Shared Memory Systems. *The Journal of Supercomputing, 15*(3), 295-320.

Young, J. W. (1974). A first order approximation to the optimum checkpoint interval. *Communications of the ACM 17*(9), 530 - 531.

Zhou, Y., Iftode, L., Sing, J. P., Li, K., Toonen, B. R., Schoinas, I., et al. (1997). Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. *ACM SIGPLAN Notices 32*(7), 193 - 205

Zhu, Y., Xiao, L., Xu, Z., & Ni, L. M. (2006). Incentive-based scheduling in Grid computing. *Concurrency and Computation: Practice & Experience, 18*(14), 1729 - 1746

Ziv, A., & Bruck, J. (1997). An On-Line Algorithm for Checkpoint Placement. *IEEE Transactions on Computers, 46*(9), 976--985.