Optimizing Checkpoint Size in the C3 System

Daniel Marques, Greg Bronevetsky, Rohit Fernandes, Keshav Pingali, Paul Stodghill
{marques,bronevet,rohitf,pingali,stodghil}@cs.cornell.edu
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

ABSTRACT

The running times of many computational science applications are much longer than the mean-time-between-failures (MTBF) of current high-performance computing platforms. To run to completion, such applications must tolerate hardware failures.

Checkpoint-and-restart (CPR) is the most commonly used scheme for accomplishing this - the state of the computation is saved periodically on stable storage, and when a hardware failure is detected, the computation is restarted from the most recently saved state. Most automatic CPR schemes in the literature can be classified as system-level checkpointing schemes because they take core-dump style snapshots of the computational state when all the processes are blocked at global barriers in the program. Unfortunately, a system that implements this style of checkpointing is tied to a particular platform amd cannot optimize the checkpointing process using application-specific knowledge.

We are exploring an alternative called automatic application-level checkpointing. In our approach, programs are transformed by a pre-processor so that they become self-checkpointing and self-restartable on any platform. In this paper, we evaluate a mechanism that utilizes application knowledge to minimize the amount of information saved in a checkpoint.

1. INTRODUCTION

The high-performance computing community has largely ignored the problem of implementing software systems that can tolerate hardware failures. This is because until recently, most parallel computing was done on relatively reliable big-iron machines whose mean-time-between-failures (MTBF) was much longer than the execution time of most programs. However, many programs are now designed to run for days or months on even the fastest computers, even as the growing size and complexity of parallel computers makes them more prone to hardware failure. Therefore it is becoming imperative for long-running scientific programs to tolerate hardware failures.

The standard technique for accomplishing this is checkpoint-and-restart (CPR for short)[1]. Most programmers implement CPR manually by (i) identifying points in the program where the amount of state that needs to be saved is small, (ii) determining what data must be saved at each such point, and (iii) inserting code to save that data on disk and restart the computation after failure. For example, in a protein-folding code using *ab initio* methods, programmers save the positions and velocities of the bases (and a few variables such as the time step number) at the end of each time step. In effect, the code becomes self-checkpointing and self-restarting, and it is as portable as the original application. Furthermore, if checkpoints are saved in a portable format, the application can be restarted on a platform different from the one on which the checkpoint was taken. To ensure a consistent view of global data structures, this approach of manual application-level checkpointing (ALC) [25] requires global barriers at the points where state is saved. Although barriers are present in parallel programs that are written in a bulk-synchronous manner [10], many other programs such as the HPL benchmark [17] and some of the NAS Parallel Benchmarks [15] do not have global barriers except in their initialization code.

A different approach to CPR, developed by the distributed systems community, is *system-level* checkpointing (SLC) [14] [7] [23] [18], in which all the bits of the computation are periodically saved on stable storage. For example, the Condor system [14], an industry standard cluster management software that provides checkpointing capabilities for sequential programs on a number of different platforms uses this technique for taking uniprocessor checkpoints. The amount of saved state can be reduced by using incremental state saving. For parallel programs, the problem of taking a system-level checkpoint reduces to the uniprocessor problem if there are global barriers where state can be saved and there are no messages in flight across these barriers. Without global synchronization, it is not obvious when the state of each process should be saved so as to obtain a global snapshot of the parallel computation. One possibility is to use coordination protocols such as the Chandy-Lamport [6] protocol.

The advantage of SLC is that unlike ALC, it requires no effort from the application programmer. A disadvantage of SLC is that it is very machine and OS-specific; for example, the Condor documentation states that "Linux is a difficult platform to support...The Condor team tries to provide support for various releases of the Red Hat distribution of Linux [but] we do not provide any guarantees about this." [9]. Furthermore, by definition, system-level checkpoints cannot be

[1]Strictly speaking, CPR provides a solution only for *fail-stop* faults, a fault model in which failing processors just hang without doing harmful things allowed by more complex *Byzantine* fault models in which a processor can send erroneous messages or corrupt shared data [13].

* This research was supported by DARPA Contract NBCH30390004 and NSF Grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140.

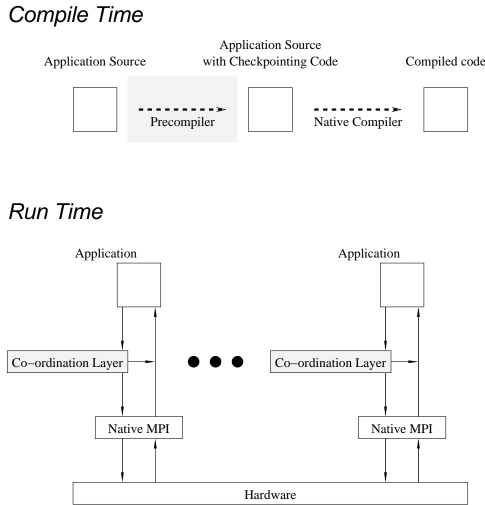

Figure 1: System Architecture

restarted on a platform different from the one on which they were created. It is also usually the case that the sizes of system-level checkpoints are larger than those produced by manual application-level checkpointing.

In principle, one could get the best of both worlds with *automatic* application-level checkpointing [2] [20] [24] [5] [21]. This requires a precompiler that can automatically transform a parallel program into one that can checkpoint its own state and restart itself from such a checkpoint. Compiler analysis is needed to identify that part of the state of the computation that needs to be saved, and code needs to be inserted to recompute the rest of the state of the computation from that saved state on recovery. To handle programs without global barriers, we need a protocol for coordinating checkpointing by different processes.

Figure 1 is an overview of our approach. The C^3 (Cornell Checkpoint (pre)Compiler) [3] [4] [5] [21] reads almost unmodified C, C/MPI and C/OpenMP source files and instruments them to perform application-level state-saving; the only additional requirement for programmers is that they must insert a `#pragma potential_checkpoint` at locations in the application where checkpoints might be taken. At runtime, when the application reaches such a location, a checkpoint will be taken if a runtime condition (such as the expiration of a timer) has been satisfied. The output of this precompiler is compiled with the native compiler on the hardware platform, and linked with a library that constitutes a *co-ordination layer* for implementing the necessary coordination. This layer sits between the application and the MPI or OpenMP library, and intercepts all calls from the instrumented application program to MPI or OpenMP.

This design permits us to implement the coordination protocol without modifying the underlying MPI or OpenMP library. This promotes modularity and eliminates the need for access to communication library code, which is proprietary on some systems. Furthermore, instrumented programs are self-checkpointing and self-restarting on any platform. The entire runtime system is written in C and uses only a very small set of system calls. Therefore, the C^3 system is easily ported among different architectures and operating systems; currently, it has been tested on x86 and PPC Linux, Sparc Solaris, x86 Win32, and Alpha Tru64, as well as MPI and OpenMP implementations from multiple vendors. This facile portability is in contrast to a typical system-level CPR system, which needs to deal with the specifics of machine architectures, operating systems, and compilers.

In this paper, we evaluate a API that utilizes application knowledge to minimize the amount of information saved in a checkpoint. We expect that a compiler analysis [12] [1] [19] can be used to automate the usage of this API, enabling ALC approaches to offer performance superior to that of SLC systems.

2. CHECKPOINTING OVERHEADS

When evaluating the performance of an automated ALC technique, it is important to examine whether the transformations performed by the ALC compiler noticeably affect the running time of the application. This overhead may come from the new code that is inserted and/or the negative effects the transformations may have on the native compiler's ability to optimize the application. In order to evaluate this we ran several benchmarks in multiple configurations:

- Original Application
- Original Application using SLC, no checkpoints taken
- Original Application transformed via C^3, no checkpoints taken.

We ran experiments comparing C^3 with Condor. This comparison was performed using *art*, *equake*, *mcf* and *cpr* from the SPEC [22] suite of benchmarks and *mg* from the NAS OpenMP[2] C suite [16] on a P4 Linux machine with a networked disk. The comparison is shown in Figure 2.

From these numbers we can see two things. First, Condor has minimal overhead when no checkpoints are being taken. This is hardly surprising since during checkpoint-free execution it needs to do little more than load up some checkpointing library code and provide hooks into the application that can be used during checkpointing. The more interesting result is the very low overhead resulting from C^3's code transformations (in order, it was approximately 0%, 1.5%, 0%, 2.5%, 1.5%).

Having established that C^3's transformations introduce minimal overhead, we need to examine the cost of taking checkpoints using C^3 vs using SLC solutions. Figure 3 shows the running time of applications that take 5 checkpoints using Condor vs their running time when they take 5 checkpoints using C^3. For the purposes of making this a fair comparison, we have written a library that limits Condor to taking checkpoints at the same potential checkpoint locations used by C^3.

Once again we see little difference between C^3 and Condor. Both take approximately the same amount of time to take the same number of checkpoints, meaning that the overhead of using C^3 or Condor in a real application would be the same for any given checkpointing frequency. Though Condor and C^3 have the same performance, a quick look at Figure 3 shows that for low enough checkpointing frequencies and large enough application state sizes both systems may have high overheads. As such, it is important to develop techniques to minimize the overheads of checkpointing.

[2]for these experiments, we ran this code sequentially

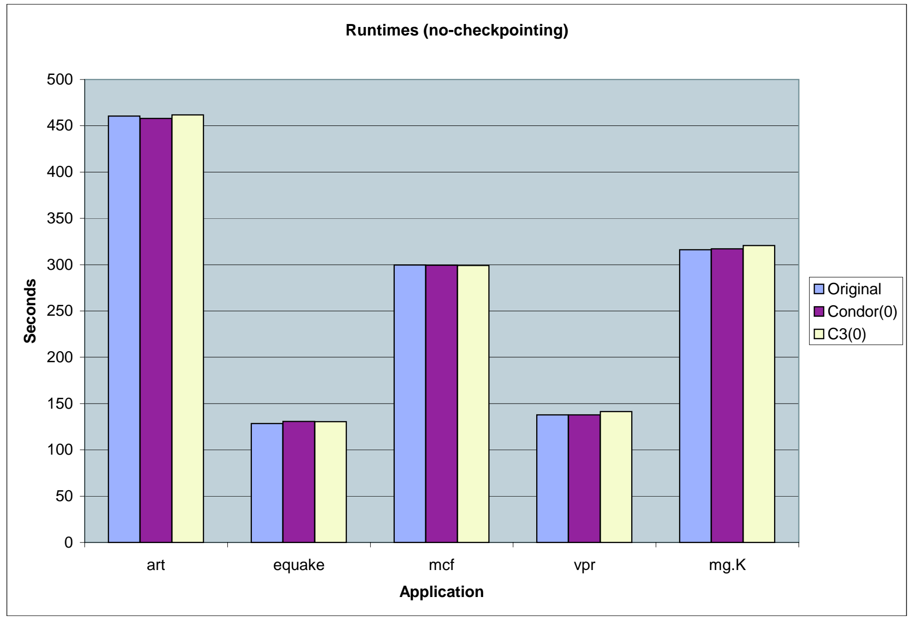

Figure 2: No Checkpoint Runtimes

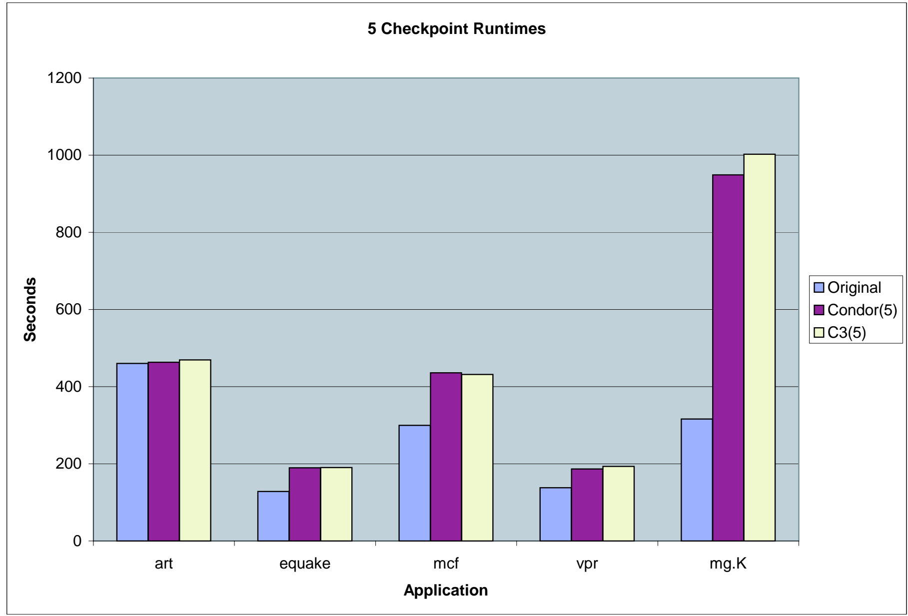

Figure 3: Cost of 5 checkpoints vs Condor

There are two orthogonal strategies for reducing the time that an application spends checkpointing: reduce the frequency at which checkpointing occurs, or reduce the amount of state that is saved at each checkpoint. Since checkpointing frequency is typically determined by environmental factors such as the MTBF of the system, for our work we assume that the checkpoint frequency is immutable and therefore focus on reducing checkpoint size. Figure 4 shows the sizes of checkpoints recorded by Condor and C^3. Two things become apparent:

- Condor and C^3 save approximately the same amount of state on all the benchmarks
- The amount of time each application spends in checkpointing is correlated with the sizes of the checkpoints

The latter point is not surprising since checkpoints are written to a network disk, which is a relatively slow resource. As such, minimizing the amount of state saved appears to be a promising way to reducing the overhead of checkpointing.

It is in this aspect that ALC and SLC significantly differ. SLC solutions know nothing about the applications they checkpoint. As such, they must save all of the address space that may be in use by the application. However, an automated ALC solution like C^3 works with the application's source code and can perform analysis to determine whether it can safely omit certain portions of application state from

the checkpoint, without compromizing recovery. Therefore, this type of analysis appears to be a promising way to minimize the overheads of checkpointing.

3. OPTIMIZING CHECKPOINT SIZE

In this section, we describe a mechanism and API for reducing the amount of state that an application needs to save at each checkpoint, and show how manually inserting calls to this API affects the checkpoint sizes and runtimes of our benchmark codes. It is our belief that these API calls could be inserted correctly and beneficially by a compiler.

3.1 A Logical Partitioning of Objects

The C^3 implementation has a portable implementation of a memory allocator that is used by codes that were transformed by the C^3 system. Our replacement allocator ensures that after restart dynamically allocated objects get "rebuilt" at the same address they had before the checkpoint was taken.

Given that we needed to construct a memory allocator from scratch, we designed our allocator to support features that would be beneficial for use in a checkpointing system. One such feature is support for logically partitioning dynamically allocated objects into *subheaps*.

The C^3 memory allocator is normally invoked via a call to `CCC_malloc()`, which takes the same parameter as a call to the C library's `malloc()` function (C^3 also provides versions of `calloc()` and `realloc()`).

As an alternative, the C^3 memory allocator may also be invoked via a call to `CCC_subheap_malloc()`, which takes an additional parameter s that specifies in the subheap this object should be placed in. The number of different subheaps is finite, and fixed at compile time. Calls to the normal `CCC_malloc()` function place the created objects in subheap 0.

Allocating objects to different subheaps represents a logical partitioning of an application's dynamic structures. Such partitioning is entirely safe, as it does not affect correctness. In fact, the partitioning is invisible to the application. Partitioning may affect performance, however, as subheaping enforces some constraints on an object's placement. Discussion on this is beyond the scope of this article. Others have studied the effect of dynamic object placement on performance [8] [11].

3.2 Specifying Subheap Checkpointing Behavior

The C^3 API for specifying how the checkpoint mechanism should treat all the objects belonging to a specific subheap at checkpoint time is the following:

- `Subheap_Save_Always(s)` specifies that at all future checkpoints taken, the system is to save the contents of subheap s.
- `Subheap_Save_Never(s)` specifies that at all future checkpoints taken, the system is **not** to save the contents of subheap s.
- `Subheap_Save_Next(s)` specifies that at the next checkpoint taken, the system is to save the contents of subheap s, but at future checkpoints after that, it only needs to save a *reference* to the most recently saved version of s.

Calls to `Subheap_Save_Never(s)` are used by the application to inform the checkpointing mechanism that, after this point, all objects of subheap s are never to be used (read-from, written-to, or `free()`-ed) again. Therefore, they don't need to be saved to the checkpoint. After this call, future `malloc()`s of an object to subheap s will succeed, but those objects will be neither saved nor restored.

Calls to `Subheap_Save_Next(s)` are used by the application to notify the checkpointing mechanism that after this point all objects of subheap s are never to be written-to again (including being `free()`-ed). Therefore, the mechanism only needs to save this subheap at the next checkpoint that is taken, C_n. At any future checkpoint, C_{n+i}, instead of saving the objects of subheap s, the mechanism will only save a reference to the version of s that was saved at C_n. If the application restarts at C_{n+i}, the objects of s will be restored as they were when saved at C_n. After this call, and after the next checkpoint is taken, future `malloc()`s of an object to subheap s will succeed, but those objects will be neither saved nor restored.

Calls to `Subheap_Save_Always(s)` specify that the checkpoint mechanism should resume saving s at every future checkpoint.

Notice that although assigning objects to subheaps always preserves program correctness, improper use of this API could result in incorrect program behavior, so the insertion of these calls must be done carefully.

3.3 Experimental Results

To evaluate the effectiveness of utilizing this API, we manually instrumented the five benchmark codes described in the previous section. For each call to `malloc()`, we specified the subheap in which all the objects built by that statement should be placed. (This is done via an insertion of a `#pragma` in the original source – the C^3 pre-compiler automatically replaces calls to `malloc()` with calls to the C^3 version of the function.) For `vpr`, we needed to identify that certain methods are object factories (i.e., a method that returns the result of calling `malloc()`, and is called by the application instead of directly calling `malloc()`) so that the pre-compiler could transform them to accept an extra parameter to be passed to `CCC_subheap_malloc()`.

We also inserted calls to `Subheap_Save_Next()` at appropriate locations in the code where they would be correct and would be beneficial for reducing checkpoint size. We then ran the codes, observing how the checkpoint size was reduced (Fig. 5), and how this affected the runtime of the application (Fig. 6) compared to the Condor and original C^3 versions of the application. We did not attempt to insert calls to `Subheap_Save_Never()` for these codes. On those charts, the version of the application using C^3 combined with this API is referred to as *C3_memex*.

For most codes, using the subheap API reduces the amount of data that needs to be saved. For `equake`, the total amount saved is reduced by 73%; for `art`, 60% for `vpr`, 50%; and for `mg`, 25%. For `mcf` there was no safe, no trivial, place to insert `Subheap_Save_Next()`, so the size of the checkpoint was not reduced.

In Fig. 6 we can see the effect that reducing the checkpoint size has on the application runtime.

- For `mcf`, where there was no reduction in checkpoint size, clearly, there was no performance benefit.

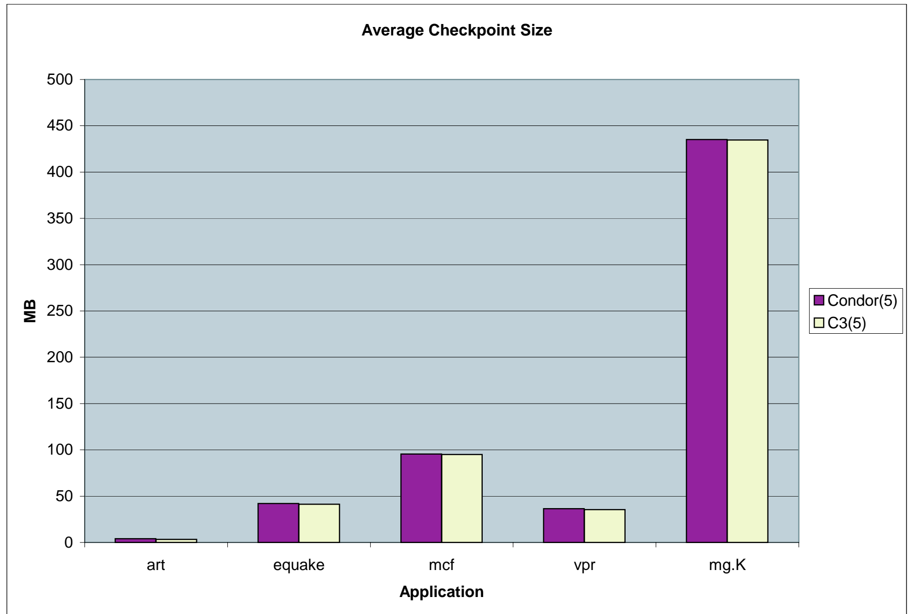

Figure 4: Checkpoint sizes vs Condor

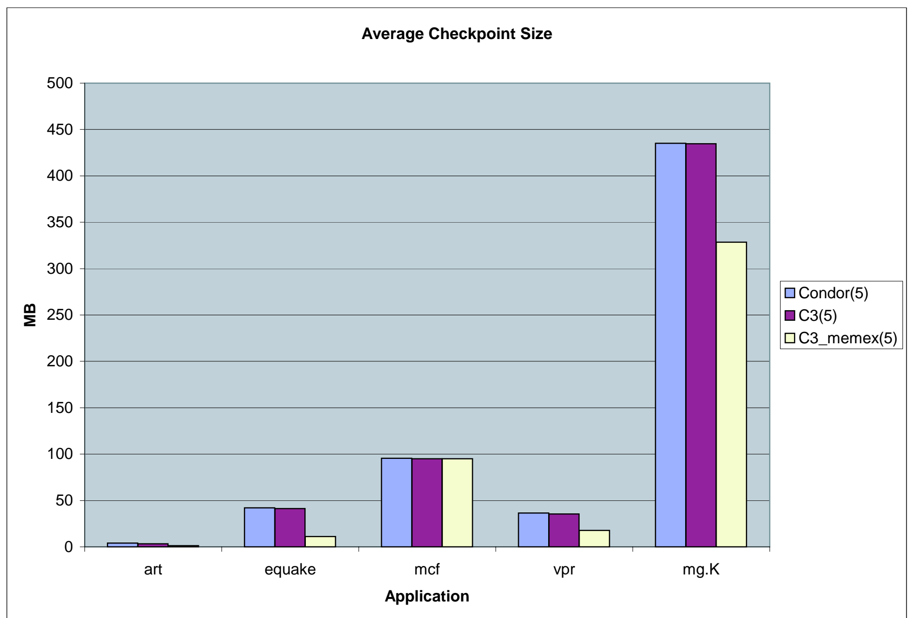

Figure 5: Average Checkpoint Size, Using API

- For `art`, where the checkpoints were very small and very little time was spent checkpointing, reducing the checkpoint size by 60% reduced the checkpointing overhead by 66% but had no noticeable effect on overall runtime.
- For `equake`, reducing the checkpoint size by 73% reduced the checkpointing overhead by 71%, reducing the overall execution time by 22%.
- For `vpr`, reducing the checkpoint size by 50% reduced the checkpointing overhead by 50% and the runtime by 14%.
- For `mg`, where the checkpoint size was shrunk by 25%, the overhead went down by 27% and the runtime by 20%.

These results show that the overhead of checkpointing is directly proportional to the size of the checkpoint. However, the benefit of state reduction on runtime depends on the details of the application.

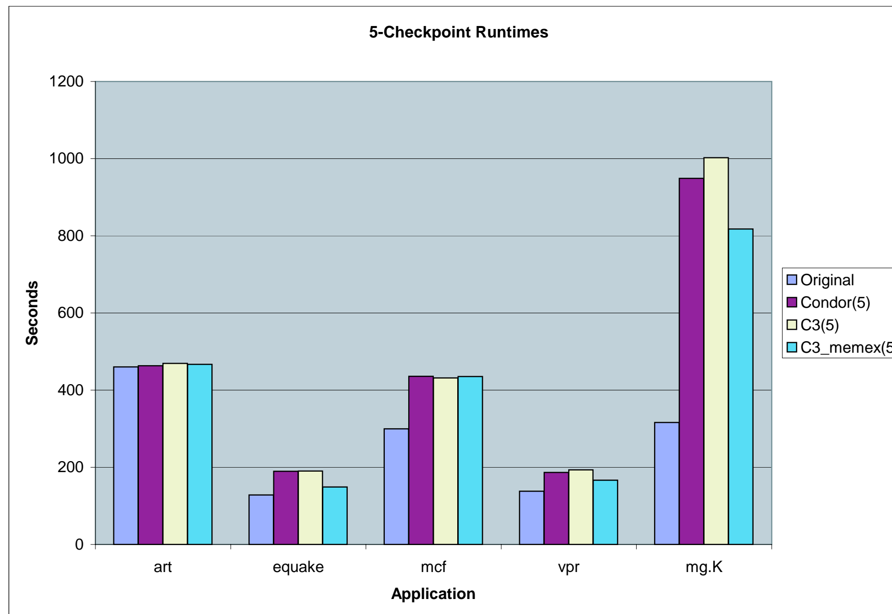

Figure 6: 5 Checkpoint Runtimes, Using API

4. CONCLUSIONS AND FUTURE WORK

We believe that these results imply that the C^3 subheap API presented is useful for reducing the overhead added by checkpointing, but that other techniques should be developed to supplement it.

We believe that the insertion of these API calls can and should be done by a compiler, because bad placement of these calls can lead to unexpected and incorrect program behavior. We are currently developing an analysis to determine the safe and beneficial spots to place such calls, and to determine a good assignment of objects to subheaps.

We envision that the C^3 system will have multiple automated mechanisms for reducing checkpoint size, and that for each application, depending on its data use patterns, a different set of these mechanisms will be applied to reduce the overhead that checkpointing adds to an application.

5. REFERENCES

[1] Micah Beck, James S. Plank, and Gerry Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, Dept. of Computer Science, University of Tennessee, 1994.

[2] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[3] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *Principles and Practices of Parallel Programming*, San Diego, CA, June 2003.

[4] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23–26 2003.

[5] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level checkpointing for shared memory programs. In *Symposium on Application Support for Programming Languages and Operating Systems (ASPLOS) 2004*, Boston, MA, October 9–13 2004.

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

[7] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. In *The IEEE Supercomputing '97*, 1997.

[8] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Conference on Programming Language Design and Implementation (PLDI) 1999*, 1999.

[9] Condor. http://www.cs.wisc.edu/condor/manual.

[10] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[11] Samuel Z. Guyer and Kathryn S. McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *OOPSLA 2004*, 2004.

[12] C-C. J. Li and W. K. Fuchs. Catch – compiler-assisted techniques for checkpointing. In *20th International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.

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

[14] J. Basney M. Litzkow, T. Tannenbaum and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[15] NAS webpage. http://www.nas.nasa.gov/Software/NPB/.

[16] NPB OMP webpage. http://phase.hpcc.jp/Omni/benchmarks/NPB/.

[17] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Available at http://www.netlib.org/benchmark/hpl/.

[18] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. Technical Report UT-CS-94-242, Dept. of Computer Science, University of Tennessee, 1994.

[19] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: optimizing the performance of checkpointing systems. *Software Practice and Experience*, 29(2):125–142, 1999.

[20] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[21] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Supercomputing 2004*, Pittsburgh, PA, November 6–12 2004.

[22] SPEC webpage. http://www.spec.org/.

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

[24] Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the Pittsburgh Supercomputing Center Terascale Computing System. In *Supercomputing*, 2001. Available at http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.html.

[25] S. Vadhiyar and J. Dongarra. SRS - a framework for developing malleable and migratable parallel software. *Parallel Processing Letters*, 13(2):291–312, June 2003.