University of Wisconsin-Madison

# Reducing the Checkpointing Burden of Condor:

# Analysis and Implementation

Computer Sciences 736

John Bent and Gregory R. Bronner

Prof. Remzi Arpaci-Dusseau

Saturday, May 13, 2000

Acknowledgements:

We wish to acknowledge the generous help of the Condor Team, and especially of the Condor Team staff members Todd Tannenbaum, Jeff Ballard, Peter Keller, Derek Wright, and Peter Couvares. The original idea to investigate Condor checkpointing came from Jim Basney, and Doug Thain and Rajesh Raman answered many of our questions. Finally, we thank Remzi Arpaci-Dusseau for his guidance and criticism of our work.

## Table of Contents:



## Table Of Figures:

## Introduction:

Condor is a distributed system that harnesses the power of users' unused workstations to deliver large amounts of computing to CPU intensive projects. Because users can and do claim their machines at unforeseeable times, Condor checkpoints programs' state periodically and migrates interrupted jobs to new host machines. Additionally, Condor checkpoints a job when it detects user activity at the terminal; this is called a *vacate checkpoint*.

As enrollment in a Condor pool is usually voluntary, the Condor system must strive to minimize user disruptions. In particular, this necessitates finding a balance between shipping as little data across the network as possible (to avoid saturating it), while vacating computers as quickly as possible. Our research represents an attempt to make Condor more user-friendly by improving vacate speeds while reducing network bandwidth.

## Background:

In UNIX, as in most modern operating systems, the state of a process consists of the contents of the memory in its address space, its open files, pending signals, the state of the registers, and any other operating system-specific features that affect the computation. The most important idea of user-level checkpointing was that it is possible to reduce the problem of saving the state of kernel-level state variables (registers, open files, etc.) to the problem of saving user-level memory by using `setjmp/longjmp` calls which have the effect of pushing kernel-type information onto the stack, where it is accessible to user-level programs, and can be written to disk. As the image size of processes has grown, fueled by the rise of virtual memory, the amount of time that it takes to write the process to disk has also grown, as I/O speeds have not kept pace with the growth in process sizes.

More recent work on user-level checkpointing has focused on making the checkpoints smaller and on improving the speed of checkpointing. The most promising idea to come out of this research effort is probably incremental checkpointing; rather than write out the entire process space to disk, these methods attempt to write out only the differences, relying on memory locality and frequent checkpointing to ensure that the differences are much smaller than the whole space.

The aim of this project was to determine to what to extent modern ideas of user-level checkpointing would improve the user and network-friendliness of Condor, without making it less stable, less architecture or operating system neutral, or less reliable. Additionally, the previous work developed checkpointing strategies that were optimized for a synthetic load of mostly numerical or mathematical software running on much slower machines and that checkpointed every few minutes. We wanted to see whether these same ideas of incremental checkpointing would be applicable in Condor, which runs a wide range of CPU intensive software and typically checkpoints much less frequently (the default interval is three hours).

## Plan of Attack:

In order to determine whether incremental checkpointing would be beneficial to Condor, we proceeded as follows:

- First we examined the relevant literature and data on checkpointing.
- Next we attempted to measure the redundancy in Condor's current checkpointing scheme by implementing a tool that did a page-by-page comparison between the current checkpoint and its immediate predecessor, and calculated the number of pages that were new and that had had data written to them (*dirty pages)*.
- We then condensed this information and imported it into a relational database, and attempted to infer whether incremental checkpointing would offer any performance improvements over conventional checkpointing. As a side-effect, we were able to test long-term patterns of memory usage in several different kinds CPU intensive programs; these results tended to agree with the published literature.
- We then set out to modify Condor to support incremental checkpointing, using some simple incremental checkpointing techniques.
- Finally, we measured the time latency performance of incremental checkpointing versus normal Condor checkpointing on a synthetic workload.

## Previous Work:

With the rise of the personal computer during the early 1980s, the aggregate amount of computing power rose quickly. At the same time, much of this power was invested in single-user workstations that spent most of their time running screen savers or waiting for their owners to return. The Condor system was developed to harness the wasted cycles from these computers, but in order to ensure forward progress, a mechanism for user-level checkpointing had to be developed.

Litzkow and Solomon developed user-level checkpointing; this evolved to work on most variants of the UNIX operating system[1]. The basic mechanism that they used was to use `setjmp` calls to capture the program state (registers, etc.) into memory, and then write the entirety of each segment (excluding the text segment, which is read only) to disk. Later, this mechanism was modified to copy the segments to a temporary segment, compress that segment using `gzip`, and send the resulting compressed file to a checkpoint server. This method is called `compressed checkpointing`; although popular in some Condor installations, it is very slow and has been turned off in many Condor pools for this reason.

More recently, a group led by James Plank developed alternative mechanisms for performing checkpointing. They reasoned that if a process does not dirty all of its pages between checkpoints, sending the clean pages back to the checkpoint server represents wasted effort, as these pages have not changed from the previous checkpoint. Specifically, they developed several techniques including:

*Incremental Checkpointing*: All writable pages are set to read only using `mprotect`; the resulting segmentation violation gets trapped; the offending page has its permissions reset to read-write and the page number is written to a list of dirty pages. This method imposes the overhead of an `mprotect` once per dirty page, as well as the time necessary to process a signal and jump to the appropriate signal handler, but can dramatically reduce the amount of data written at checkpoint time if a significant number of pages are not modified between checkpoints.

*Copy-On-Write Checkpointing*: As dirty pages are swapped out of memory, they are written to a log-structured checkpoint file; at checkpoint time only dirty pages must be written out to disk. This has the advantage of reducing vacate time, but the disadvantage of requiring access to the kernel paging mechanism.

*Bitwise Compressed Differences:* At checkpoint time, the dirty pages are `XOR`ed with the previous contents of the page. In the event that the pages are similar, most of the contents of the difference will consist of long sequences of zeros. In that case, the differences can be compressed using standard compression algorithms, and the result will be much smaller than one page.

*Human Assisted Checkpointing:* Using information supplied by the user, in the form of `include_bytes, exclude_bytes,` and `checkpoint_here` calls, the amount of data to be checkpointed can be reduced dramatically by eliminating the checkpointing of memory whose value is no longer needed for the computation, and by choosing to checkpoint at times when the total amount of data to be checkpointed is least (e.g. at the beginning of an iteration for an iterative process). We chose not to use these ideas because they weren't applicable to the research that we were performing, and because we didn't really believe that most people were willing or able to modify their programs in order to run them under Condor.

*Compiler Assisted Checkpointing:* Basically this is the same as human assisted checkpointing, but it involves using the compiler to make the judgments about when to checkpoint. Although this is a promising technique and may well offer large savings, it was beyond the scope of this project and somewhat outside of our areas of interest.

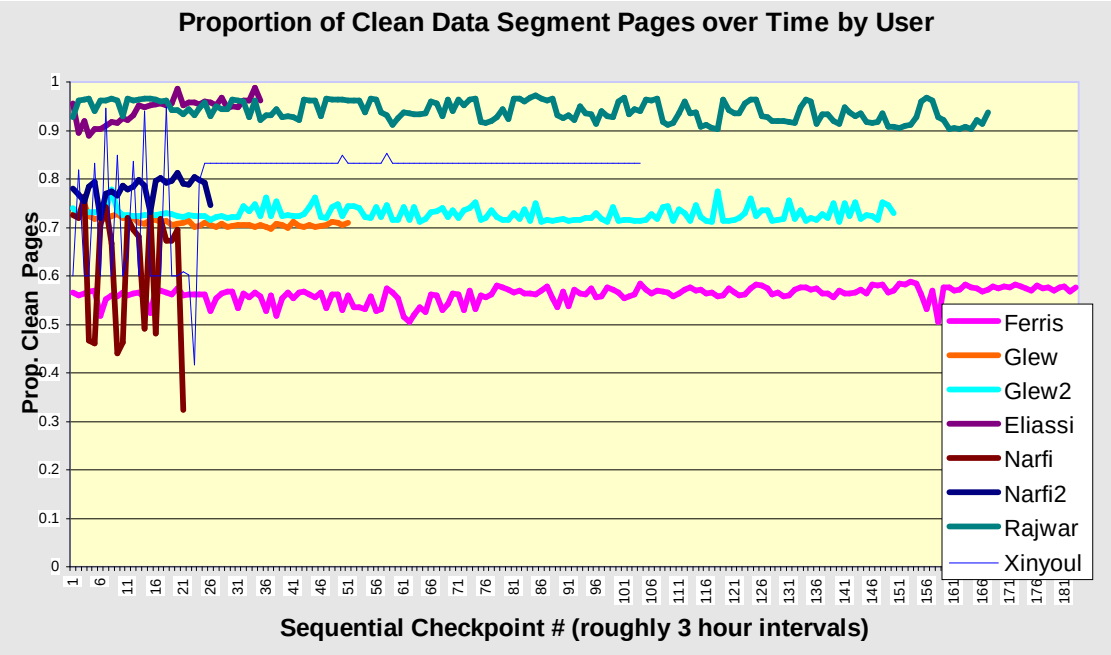
Plank's group validated their results through a series of synthetic benchmarks consisting primarily of publicly available mathematical or computational physics software. What they found was that over a ninety second interval, the efficiency of various checkpointing mechanisms depended upon the rate at which the pages were dirtied; in cases where most pages were not dirtied, incremental checkpointing was a clear win.

**Figure 1: Most users' programs are consistent in the percentage of their data segment pages that they update.**

## Other Checkpointing Ideas:

*Probabilistic Checkpointing:* The idea behind this was that by hashing the value of all memory in a page, we could obtain a piece of summary information which could be compared with a page on disk or across the network easily. The trouble with this mechanism is that unless the hash code is as long as the page, there is no way to

[1] While versions of Condor have been developed for non-UNIX operating systems, all have been shipped in a 'clipped wing' format, meaning that they do not support checkpointing.




determine whether the pages are identical. Through mathematics, we could guarantee that the probability of determining two different pages to be the same would be less than an arbitrary number, albeit at increasing expense.

Like lottery-based scheduling, probabilistic checkpointing failed to gain wide-spread acceptance because of most people's reluctance to accept anything other than deterministic methods. Indeed, Professor Miron Livny, head of the Condor project, reacted quite strongly against it, correctly arguing that potentially inaccurate computation would be hard to sell, and that many users would be justifiably reluctant to use such a system.

## Our Work:

Our work is composed of two parts. In the first section we designed and implemented a system to measure the dirtiness of pages in processes submitted to the Condor system. In the second part, we implemented an incremental checkpointing system, and compared the performance of this system to the performance of the standard sequential checkpointing algorithm on a synthetic workload.

## Part One: Measurement of the Dirtiness of Pages in Long Running Processes:

We measured the dirtiness of user jobs by adding code at the checkpoint server. We did this by examining the checkpoint files immediately after the checkpoint, and comparing the most recent checkpoint file with its predecessor. In order to verify the accuracy of this process, we wrote programs that dirtied specific patterns of bytes and then checkpointed the image; the resulting summary of the checkpoint file was visually inspected for accuracy.

The checkpoint file format is fairly simple; it consists of a header, a number of segment information structures, and the a series of data blocks which correspond to the data contained in each segment. Using this information, were able to collect statistics about the process on a page-by-page basis; at checkpoint time a script appended the most recent statistics to a log file stored on the checkpoint server.

Specifically, the statistics that we kept consisted of the size of the process, the page size of the host architecture, the number of dirty pages, the percent of modified bytes on the dirty pages, the number of new pages added to the process, the number of segments and their types, the owner of the job, its name, and the time of the checkpoint. Although we did collect the data necessary to determine the page locality of dirty pages (e.g. the history of a specific page over time), we did not incorporate this data into our database, as it would have made our database too large; a future study may wish to explore this more fully.

Because of the necessity of not overloading the checkpoint server, we did not attempt to perform analysis at the time of data collection. Additionally, none of the users were notified that we were running this checkpoint analyzer during the investigation period, nor did Condor's speed or reliability change noticeably during the period; consequently we believe that our measurements did not affect the quantities measured, nor did it induce users to behave differently during the measurement period than they do at other times.

During the measurement interval, which lasted approximately one month, we checkpointed over 1800 jobs, comprising the work of nine distinct users, and consisting of over 1.6 terabytes of data. After collecting the data, we were careful to remove the checkpoints generated by members of the Condor team for testing and development purposes.

The first result that we obtained was that in long-running jobs, the size of the data segment dominates the size of all other writable segments combined. Most shared library segments are quite small, often consisting of only a page or two of memory, and the stack segments rarely exceed five or six pages; about 98% of the pages checkpointed were in data segments. After realizing this, we concentrated our efforts on reducing latency in checkpointing the data segment; the potential benefits to improving the checkpointing of other segment types did not justify the implementation overhead.

Next we observed that the proportion of clean pages in each checkpoint stayed relatively constant when viewed on a per user basis. This is shown in figure 1; note that for most users the proportion of clean pages varied only slightly over time.

More importantly, this showed us that the past history of a process' propensity to modify memory was a fairly reliable indicator of its future propensity to do the same; for some users the correlation between the percentage of clean pages at time `t` and at time `t+1` was as high as .91; the average value was greater than 0.7, meaning that the previous value predicted at least 70% of the variance in the observed value. n which user's job was being checkpointed. This result can be most easily explained by the fact that must users ran the same programs multiple times, only varying the input data.

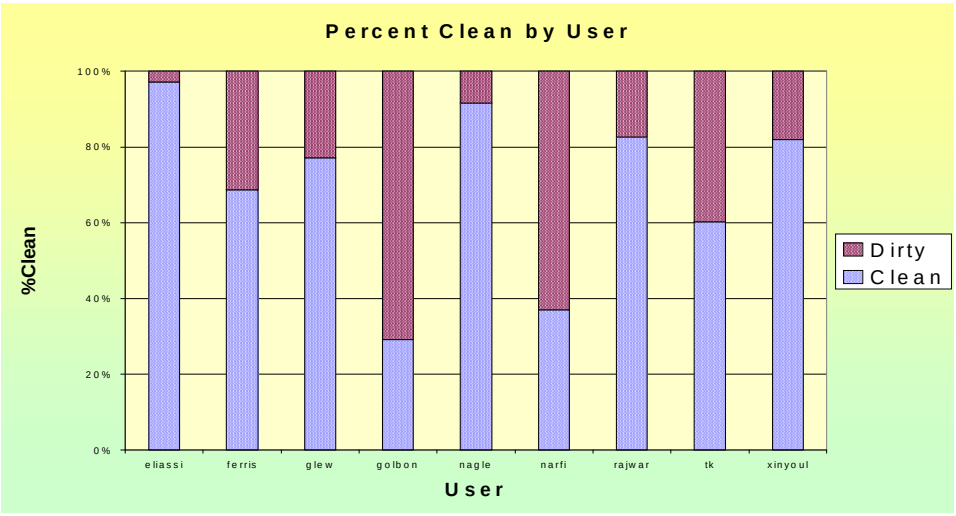



**Figure 2: Percentage of clean pages by user. This is highly user dependent because different users run different kinds of jobs. Also note that these are generally higher than synthetic benchmark figures generated five years ago. [Not shown: The standard deviation of these averages was very low.]**

The most interesting aspect of figure 2 is that the percentage of clean pages can be quite high, and is much higher than the Plank group reported, even though Condor checkpoints a process every three hours, while the Plank group checkpointed their programs every ninety seconds. We have several explanations for this discrepancy:

1. The Plank group ran primarily mathematical simulations which tended to perform random access on matrices; several of these synthetic benchmarks actually touched almost all of their memory because they performed matrix computations of the form $A_{n+1}=F(A_n)$; because this type of computation writes a new matrix over the old one, it tends to touch many pages.
2. Condor users have been moving away from using Condor solely for numerical calculation; they now tend to submit other types of jobs including artificial intelligence simulations (eliassi), integer programming (ferris), integer benchmarks (SPECint 95) running on a simulated processor at low speed (glew), numerical simulations (narfi), etc.. Some of these jobs may be sticking large data structures into memory and just reading from them.
3. Object oriented programming tends to create large objects, but much of the object doesn't change much after the object is initialized; this would create pages that were perennially clean.
4. Certain types of simulation codes have become more efficient by emphasizing computation only in areas in which the investigator is interested. An example of this would be a climate simulation that confined most of its updates to North America[2].

[2] Andy Glew suggested this idea.

5. Using the working-set model of program behavior, a program may rapidly dirty most of the pages that it is interested in, but confine all successive writes to those pages; this could be tested by running the checkpoints more often, but we have not had time to do this yet.

Using this data, we were able to calculate the potential benefits of incremental checkpointing. They are summarized in Figure 3.

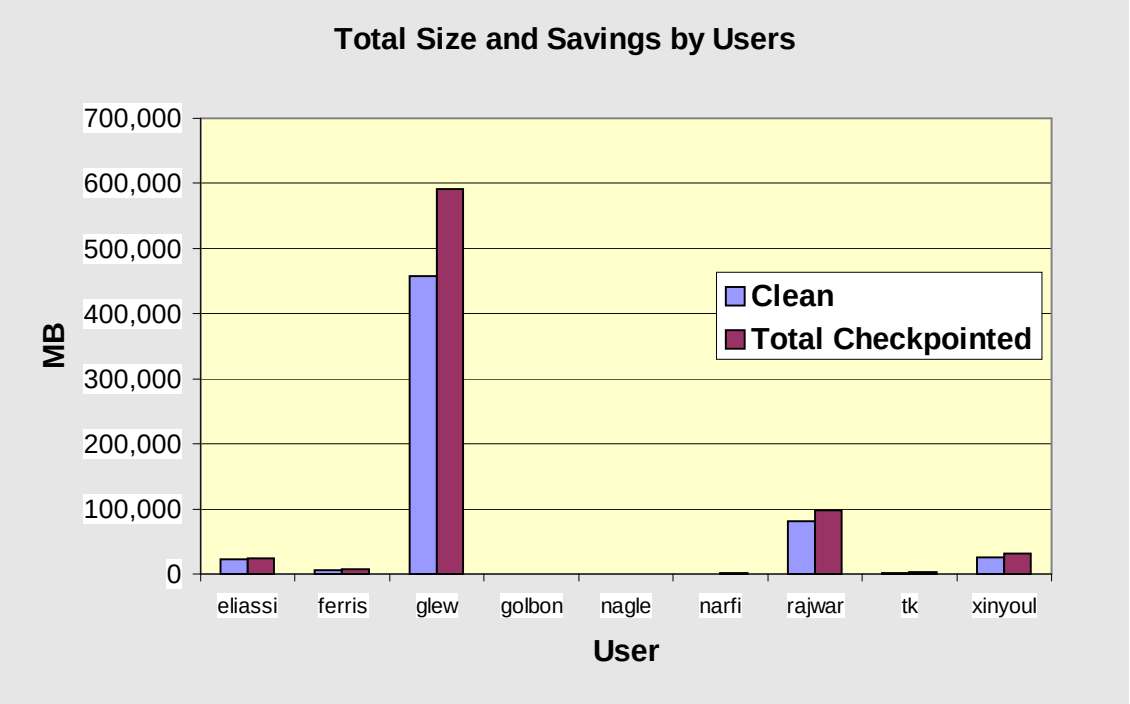



**Figure 3: Total potential savings and total data checkpointed. Note large usage differences between users.**

What is readily apparent is that a significant amount of data that was checkpointed consisted of unchanged pages. Over the 23 days that we kept statistics on unchanged pages, we determined that roughly 0.7 of the 1.2 terabytes that we checkpointed were clean pages. This represents a significant opportunity to reduce the latency and overhead of checkpointing and reduce by about 60% the Condor network traffic. In the current UW pool, this would result in a savings of 30 gigabytes of network bandwidth every day. Based upon this data we decided to continue implementing incremental checkpointing.

Additionally, Figure 3 also confirms a historical fact of Condor usage: the top few users typically account for almost all of the resource consumption. In particular, during our experimental period, several users ran long-running jobs that checkpointed often, while others, such as the user `tk` dispatched hundreds of short-lived jobs that often finished in three hours or less, thus obviating the need for checkpoints.

## Part Two: Implementation of Incremental Checkpointing:

Once we had established the network bandwidth savings that could be achieved with a policy of incremental checkpointing, we were curious to examine whether we could expect a similar reduction in the time latency of the checkpoint event. As mentioned before, Condor jobs often must vacate a non-dedicated execution machine when keyboard activity at that machine indicates that the machine's user has returned. In order to be as sociologically non-intrusive as possible, Condor must either checkpoint very quickly or lose all throughput since the last periodic checkpoint[3].

An implementation of incremental checkpointing uses a Unix system-call, *mprotect*, to write-protect the

[3] Jim Basney, a researcher at the University of Wisconsin, has recently been working on a third option, "slow" checkpointing. Slow checkpointing splits writing the checkpointing image into multiple writes and sleeping between them to lower the user's perception of Condor's intrusion.

image's pages in memory and a signal handler function to respond appropriately to the resulting segmentation violations that occur when the user code attempts to write to the now protected memory. By mprotecting the image following either a checkpoint or a restart event, we can use the resulting segmentation violations to track which pages of an image have been modified since the previous checkpoint. Subsequent checkpoints need only write this set of dirty pages.

From our measurements, we had seen that 98% of the pages in processes are contained within the data segment (i.e. the stack and the shared libraries contribute an insignificant number of pages to the total image size). For this reason, we decided to implement incremental checkpointing on the data segment only and retain the current policy of sequential checkpointing for the other segments within the memory space of the process.

Within the signal handler function, we wrote code to reset the permissions on the offending page so that the violating write instruction can successfully complete when execution returns from the signal handler. Before we can return from the signal handler however, we must update a data structure somewhere to remember which pages have been modified. One somewhat tricky aspect of the implementation is the placement of this data structure. Situating it within the data segment, which is the default location for dynamically allocated memory, will result in re-entrant signal handling when we attempt to modify it. This can have disastrous consequences.

To avoid this, we maintained our data structure in a new segment of memory which we allocate for this purpose by *mmaping* /dev/zero[4] to an unused space in memory. We allocate this segment to hold a few variables for accounting and to remember the original end address of the data segment and to hold a bitmap containing one bit for each page in the data segment. The original end address of the data segment is used to compute whether the data segment has grown since the last checkpoint. This allows us to include any new pages with the modified pages when we write our incremental checkpoint.

To measure the overhead of adding *mprotect* and the signal handling of the segmentation violations, we ran two microbenchmarking programs in three different operating systems. One program microbenchmarked the *mprotect* system-call, by allocating a 10,000 page array and then *mprotecting* each page twice, first as read-only and then as read-write. The other program microbenchmarked our signal handling code by similarly allocating a 10,000 page array and then *mprotecting* each page as read-only and then triggered our signal handler by attempting to write to that page. Each job was then submitted one hundred times to the UW Condor pool to elicit realistic behavior.

| | | Linux, X86 | Solaris, SPARC | Solaris, X86 |
|---|---|---|---|---|
| **Mprotect** | Avg. | 2.98 us | 5.20 us | 5.16 us |
| **Signal Handling** | Avg. | 22.89 us | 167.76 us | 110.36 |

**Figure 4: The overhead of incremental checkpointing.**

Our microbenchmarks show that the overhead of incremental checkpointing is minimal. For example, even if every single page in a 10,000-page image is dirtied between checkpoints, the overhead of handling the segmentation violations and the *mprotect* call for each page adds less than two seconds of execution time between checkpoints. Given Condor's current three hour interval between periodic checkpoints, this results in less than 0.02% overhead even in the most pathological case.

To measure any reductions in the checkpoint time latency, we wrote a program that allocated a large array and purposefully manipulated its pages and explicitly called the checkpoint routine within the Condor code with which it was linked. To show the behavior within all possible ranges of locality, we called checkpoints after both modified pages using randomized locality and then using clustered locality. Our hypothesis was that the locality of the dirty pages would have no effect on sequential checkpointing which always writes the entire image but that incremental checkpointing would fare much better when pages are clustered. Since incremental checkpointing initiates a system-call for each clump of modified pages, randomly scattered modified pages should take longer to incrementally checkpoint than if that same number of pages had clustered locality.

[4] /dev/zero is a special file in *nix which appears to contain an infinite amount of zeros when read and discards any data which is written to it.

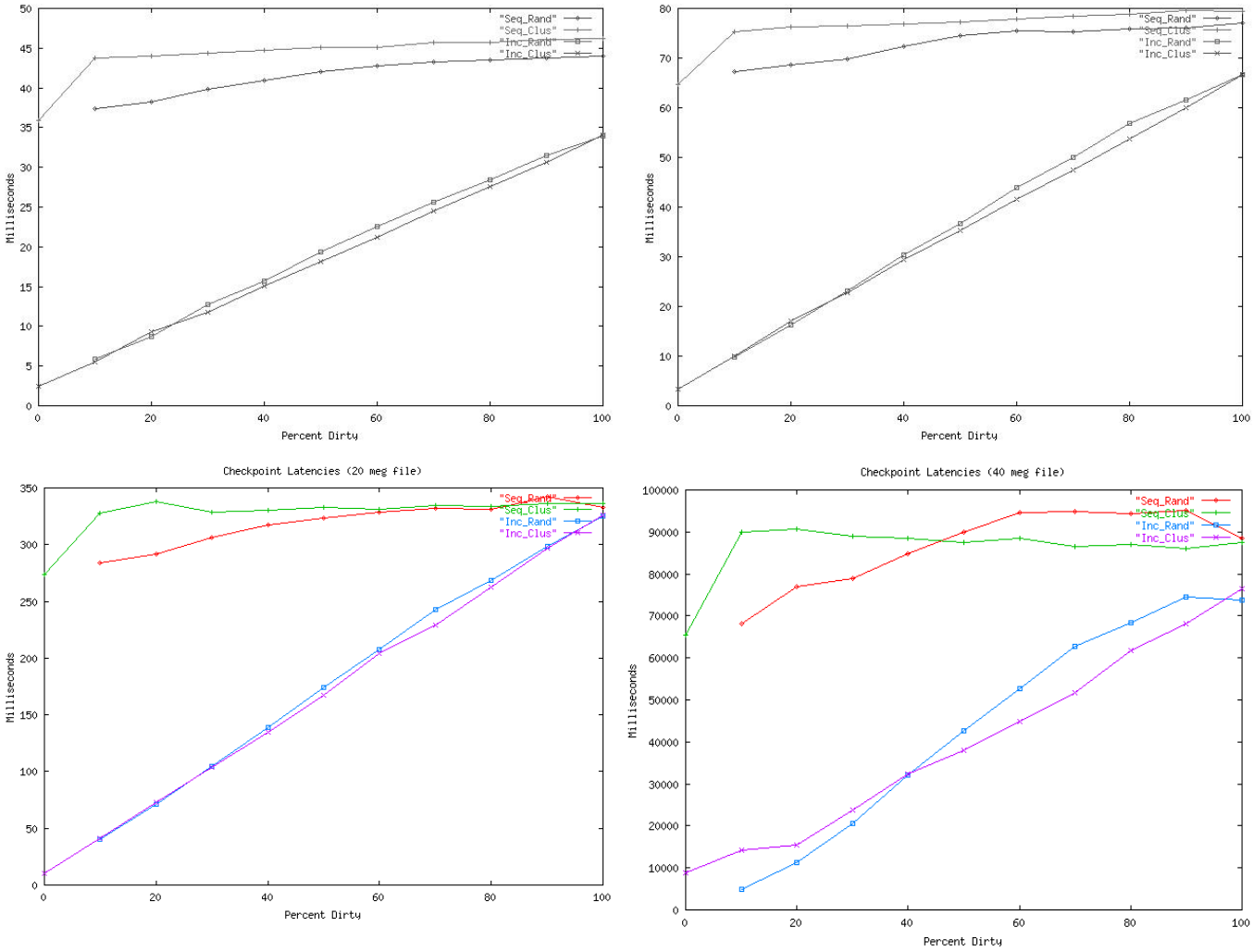



**Figure 5: Checkpoint latency times as a function of image size and percent and locality of modified pages.**

One seeming anomaly is that incremental checkpointing is still faster than sequential checkpointing even at 100% dirty. In this case, all pages will be written and incremental checkpointing should be slower because it has to examine the bitmap. The reason that this is not the case in our measurements is because the 100% refers to the pages which were actively in the control of the program. Our test code dirty every single page of the array which it had allocated. However, the array did not fill the entire data segment. Clean portions of the data segment outside of our array were not checkpointed by incremental checkpointing but were by sequential checkpointing. Regardless of the total image size, these clean pages outside of our array were usually about 200 pages; they consist of pages allocated by the C runtime library and other libraries; in any event they don't affect the measurements much.

## What We Learned and What We Would Do Differently:

Unlike some groups, we picked a project that allowed us to work independently, had clear milestones, and most importantly, allowed us to view preliminary results very quickly. During the course of the project, we worked independently and at different times, communicating through email. As each of us had a different skill set, this allowed us to occasionally work at problems in parallel without stepping on each others toes. Having clear milestones allowed us to focus on near-term objectives, and encouraged us to focus on completing small sections of the project without feeling as if we had to get everything done all at once; it also gave us a sense of where we were relative to our original estimates. Finally, picking a problem that allowed us to start generating data quickly allowed us to avoid the trap of having to complete the project before obtaining any data, and allowed us to use preliminary analyses of the data to steer the project in more promising directions.

Initially we concentrated on trying to re-use existing code, but we were unable to separate the classes from the libraries in which they were embedded; this resulted in some very simple executables that were over 2 mb in size, and were excruciatingly slow. Our further attempts were stand-alone programs; we re-implemented a lot of the data structures and made them independent of the Condor libraries. As a result, we had to learn UNIX system calls, signals, etc., and became quite familiar with the man pages; as a result of reading many of the header files in

`/usr/include` on several different OS/OS version combinations, we gained a newfound understanding of how an operating system works. We also benefited from the expertise of the Condor team in figuring out the differences between similar system calls and communications paradigms.

One task that occupied a lot of time was attempting to get our checkpoint comparison tools to work with every possible OS, OS version, architecture, and compression level of checkpoint and checkpoint server. In particular, obtaining accounts on the respective machines, accumulating a testing library of representative checkpoints, and ensuring that the code would be not only cross-platform, but cross-architecture as well proved to be quite tedious. When confronted with a user who was working on a 64 bit machine, we elected to stop development on our tools and simply ignore his checkpoints (which were only 40 out of 20,000). If we had to do it again, we would prefer to work in a more homogenous environment, as system calls and signals tend to be some of the least portable portions of a program.

We managed to start early enough to collect a significant amount of data, but we collected the data before we designed the database to analyze it. This resulted in us having to write, debug, and test a series of custom parsers; had we started analyzing data immediately after collecting it, we would have been able to design our log files to be more easily interpretable by the database, we would have gotten more complete data, and we would have been able to omit a long series of steps necessary to fix incomplete row entries in the database[5].

We would have also liked access to a higher end database on a more expensive piece of hardware, as joins against a large table are extremely slow in Access (they sometimes took well over half an hour on Greg's laptop), and our tables comprised around 35 MB without accounting for indexes, and the additional columns that we added.

Finally, we would have liked access to a lot more disk space and/or a really large tape array to archive checkpoints as they came in; this would have allowed us to do more analysis, would have allowed others to check and validate our data, and would have supplied a series of trace files for future checkpointing investigations.

## Conclusions:

Based upon our investigations, we believe that incremental checkpointing is worthwhile for many non-numerical jobs submitted to the Condor system. We have seen that the overhead of performing incremental checkpointing is negligible, the reduction in vacate latency and network bandwidth significant, and in the worst case, incremental checkpointing remains comparable to sequential checkpointing. Furthermore, because of the similarity of users' jobs and because of jobs' propensity to touch roughly equal fractions of pages each checkpoint cycle, we have two powerful mechanisms for determining which jobs would be good candidates for incremental checkpointing.

The major reservations that we have about incremental checkpointing are that it adds complexity to the checkpointing code, that it adds one more system call (`mprotect`) to the list of calls that are not supported in Condor, and that it is somewhat less portable across operating systems. However, we believe that a properly abstracted, architecture-independent checkpointing library could make Condor a much more network-friendly product. There is also the potential issue of overwhelming the checkpoint server. Since the checkpoint server has to update the checkpoint file with the new data, it might not be able to respond as quickly to checkpoint requests. We personally doubt that this would be a problem since we reduce the server workload by reducing the amount of data which it receives, but if it is a problem, a site could choose to establish multiple checkpoint servers. Finally, there is a very slight risk of incorrect results due to the non-idempotence of the checkpoint file update process using incremental checkpointing; if half the blocks in the file have been updated and the checkpoint server crashes, the checkpoint is in an inconsistent state, and cannot be used to restart the process.

Additionally, the benefit of having the ability to store a series of checkpoints could be quite invaluable for debugging purposes, or for restarting long-running distributed computations in the face of hardware or software failure that occurred before the previous checkpoint.

[5] As an example, we didn't start collecting page size information until about a week after we started, so we had to go through, find all entries without page size information, identify the computer that the checkpoint came from, find another entry from that computer with a correct page count, and update all incorrect rows. As a last resort, we had to email the CSL and ask for information about a particular computer.

## Future Work:

Some Condor installations use compression. Due to the sociological intrusion of a slow vacate checkpoint, the UW pool does not currently compress its checkpoints. However we would like to see how a policy of compressing the sequential checkpoints compares to incremental (or even compressed incremental ) checkpoints, especially with respect to network usage, checkpoint storage requirements and vacate latencies.

Next we would like to be able to set up a synthetic checkpointing workload, so that we could compare various approaches to checkpointing on the same data. This could either take the form of a set of programs that periodically generated checkpoints, or, in the event that we can find a terabyte of disk or tape, a trace of a month of Condor checkpoints.

Finally, we would really like to expand the number of users in the Condor user pool, as it seems premature to re-architect the checkpoint code and checkpoint server on the basis of observations of the jobs of nine users. This could be done by either expanding the number of UW Condor pool users, or by running our measurement software at other sites[6].

[6] We tried to do this at New Mexico and INFN, but ran into organizational and technical problems.

## BIBLIOGRAPHY

Michael Litzkow and Marvin Solomon, (1992). “Supporting Checkpointing and Process Migration Outside the UNIX Kernel,” USENIX Winter 1992, (283-290).

Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny, “Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System”

James Plank, Micah Beck, Gerry Kingsley, and Kai Li, (1995). “Libckpt: Transparent Checkpointing under UNIX,” *USENIX Winter 1995 Technical Conference*

James Plank, Kai Li, (1994). “Low-Latency, Concurrent Checkpointing for Parallel Programs,” IEEE Transactions on Parallel and Distributed Systems, 5(8), Aug 1994, pp 874-879

James Plank, Yuqun Chen, Kai Li, Micah Beck, Gerry Kingsley, (1995). “Memory Exclusion: Optimizing the Performance of Checkpointing Systems,” Technical Report UT-CS-96-335, August 1996

James Plank, Jian Xu, and Robert Netzer. (1995) “Compressed Differences: An Algorithm for Fast Incremental Checkpointing”, Technical Report CS-95-302, University of Tennessee