Compressed Differences:

An Algorithm for Fast Incremental Checkpointing

James S. Plank †

Jian Xu ‡

Robert H. B. Netzer ‡

† Department of Computer Science
University of Tennessee
Knoxville, TN 37996
`plank@cs.utk.edu`

‡ Computer Science Department
Box 1910
Brown University
Providence, RI 02912
`[jx,rn]@msr.epm.ornl.gov`

August, 1995

Technical Report CS-95-302
University of Tennessee

Available via `ftp` to `cs.utk.edu` in `pub/plank/papers/CS-95-302.ps.Z`
Or on the web at `http://www.cs.utk.edu/~plank/plank/papers/CS-95-302.html`

Submitted for publication. See the web page for futher information.

Compressed Differences: An Algorithm for Fast Incremental Checkpointing

James S. Plank †* Jian Xu ‡ Robert H. B. Netzer ‡

†Department of Computer Science, University of Tennessee
‡Department of Computer Science, Brown University

August 22, 1995

Abstract

The overhead of saving checkpoints to stable storage is the dominant performance cost in checkpointing systems. In this paper, we present a complete study of *compressed differences*, a new algorithm for fast incremental checkpointing. Compressed differences reduce the overhead of checkpointing by saving only the words that have changed in the current checkpointing interval while monitoring those changes using page protection.

We describe two checkpointing algorithms based on compressed differences, called *standard* and *on-line* compressed differences. These algorithms are analyzed in detail to determine the conditions that are necessary for them to improve the performance of checkpointing. We then present results of implementing these algorithms in a uniprocessor checkpointing system. These results both corroborate the analysis and show that in this environment, standard compressed differences almost invariably improve the performance of both sequential and incremental checkpointing.

Keywords: Checkpointing, rollback recovery, compression, fault-tolerance, playback debugging.

1 Introduction

Checkpointing is the act of saving an intermediate state of a program to stable storage so that it may be resumed at a later time. It is a general technique that has been used in many applications including fault-tolerant computing [1, 2], program debugging [3, 4] and others [5, 6, 7]. Checkpoints are used for fault-tolerance in a straightforward manner: the programmer periodically checkpoints the execution of his or her program so that following a failure, the program may be restarted from the most recent checkpoint, thus minimizing the amount of lost work. Checkpoints are used in debugging when a programmer needs execution

*James Plank is supported by NSF grant CCR-9409496 and the ORAU Junior Faculty Enhancement Award. Robert Netzer and Jian Xu are supported in part by NSF grant CCR-9309311

replay to track down bugs in a long-running program. In such cases, the programmer can checkpoint the program periodically and then replay its execution from the various checkpoints to help track the bug.

A checkpoint typically consists of the contents of an application's address space and its register state. The major overhead involved in checkpointing is writing the checkpoint to stable storage. As computers are being manufactured with up to 100 Mbytes of physical memory, checkpoints can be rather large, and simple schemes that save complete checkpoints at each invocation can take anywhere from a few seconds to tens of minutes per checkpoint. While this overhead may be tolerable for some checkpointing applications, it imposes too great a penalty on running time or on the checkpointing interval, and can be improved.

Several approaches have been studied to improve the overhead of checkpointing. Techniques include incremental checkpointing [3, 8, 9], compression [10, 11], buffering [12], copy-on-write [9, 12], compiler assistance [10] and diskless checkpointing [13].

This paper presents a new technique to improve the performance of checkpointing called "compressed differences." This technique can be viewed as a combination of incremental checkpointing, buffering, and very fast compression. The novelty of compressed differences is that they enable the checkpointer to trace incremental checkpoints at the word level while instrumenting the program at the page level. The result is a smaller checkpoint size, which translates to a reduction in checkpointing overhead.

We analyze the theoretical performance of this algorithm, deriving the conditions under which it improves the performance of checkpointing. We have also implemented this algorithm on a Sparcstation 2 under SunOS 4.1.3 and conducted several experiments. The results of these experiments show that the conditions for improvement are usually met: compressed differences are almost invariably an improvement over simple and incremental checkpointing, lowering checkpoint overhead in terms of both space and time.

This algorithm can be applied wherever incremental checkpointing can be applied, and should improve checkpointing overhead on uniprocessors, multiprocessors, multicomputers and distributed systems. Moreover, it exploits the trend of increasing processor speed and physical memory size to cope with the problem of limited disk bandwidth. As the disparity between processor speed and disk bandwidth continues to widen, the benefits of compressed differences should become greater over time.

2 Background and Related Work

In this section we review previous work on incremental checkpointing and compression for reducing the overhead of checkpointing. For a more detailed discussion of other techniques, we refer readers to the Ph.D. theses by Elnozahy [14] and Plank [15].

2.1 Incremental Checkpointing

A checkpoint of a single process is usually composed of the process's address space and the state of its registers. Simple checkpointers (called "sequential checkpointers") save the entire address space at each checkpoint. *Incremental* checkpointers reduce the amount of information saved at each checkpoint [3, 8, 9, 16]: only pages modified since the last checkpoint are written out to disk. Pages that have not changed since the last checkpoint can be retrieved from previous checkpoints.

On some systems, the set of pages that have been modified within a checkpoint interval may be determined by a system call (such as `pagemod()` in IGOR [3]). Most operating systems, however (e.g. most Unix systems) do not support such a call, but its effect can be achieved by using the virtual memory access protection facility [3, 17]. The checkpointer write-protects all of a process's pages at the beginning of an interval. When the process tries to modify a write-protected page, a protection violation is generated and caught by a user-mode fault handler. The handler adds the page number of the faulting address to a list of changed pages and turns off write protection for that page. At the end of a checkpoint interval, the changed page list contains all the pages that have been modified within the interval.

One problem with traditional incremental checkpointing is its page-level granularity. If only a few words within a page have been modified, then it is more efficient to store those words directly instead of the whole page. Netzer and Weaver have developed a strategy for tracing long-running programs based on these observations [18]. They instrument program executables to catch changes at the variable-access level instead of at the page-access level. Their experiments show that this strategy is effective at reducing the amount of information per checkpoint, especially when checkpoint interval is short (e.g. to support program debugging). The overhead of this scheme is much lower than expected: program executions are penalized by a factor of 1.7–2.0. However, this still represents too much of a slowdown for all but the most serious debugging tasks, and is not yet a feasible strategy for general-purpose checkpointing.

2.2 Compression

Data compression has been used for a wide variety of purposes, from I/O bandwidth reduction [19] to file system efficiency [20, 21] to extending physical memory [22]. The basic idea in all these systems is to exploit the structure of the data and trade off CPU speed for disk or network bandwidth. Compression has also been shown to be effective at reducing checkpoint size [10, 11]. Instead of writing the process state directly to stable storage, the checkpointer first compresses it. However, the potential benefits of compression for reducing the overhead of checkpointing depend on the speed of compression as well as the compression ratio. Plank and Li have shown that compression only reduces the overhead of checkpointing when

$$F_{comp} > \frac{R_{disk}}{R_{comp}},$$

where R_{disk} is the rate of disk writes, R_{comp} is the rate of compression, and F_{comp} is the compression factor [11]. Accordingly, standard compression algorithms like LZW [23] and Wheeler's [21] are too slow to improve the performance of checkpointing unless the ratio of processor speed to disk speed is very high (as in, for example, a multicomputer system like the iPSC/860 where there are up to 128 processors and just four disks [11]).

2.3 Where Our Algorithm Fits In

In this paper, we use a much faster algorithm to compress checkpoints. The algorithm stems from the diskless checkpointing work of Plank and Li [13]. The insight of this algorithm is that if we save the changes from the previous checkpoint instead of the values of the new checkpoint, then we can quickly compress away all words that are unchanged from the previous checkpoint. The result is a checkpointer that instruments programs at the page level, but uses compression to save changes at the word level. This compression is very fast, which means that even small compression factors (as low as 0.16 in our experiments) can benefit the performance of checkpointing.

We present this algorithm in the context of uniprocessor checkpointing. It should be obvious that the algorithm may be employed whenever incremental checkpointing is applicable. Moreover, its utility grows as processor speed increases relative to disk speed, and thus should be extremely useful on parallel and distributed platforms.

3 The Checkpointing Algorithm

We assume that we are checkpointing a uniprocessor using standard incremental checkpointing on a Unix-like system (i.e. no `pagemod()` system call). We treat the beginning of the program as checkpoint zero. Immediately after each checkpoint (including checkpoint zero), we protect the address space of the program to be *read-only*. Every time the program attempts to write a *read-only* page, a page fault is generated and caught by the checkpointer, which puts the faulting page number into a changed page list, reprotects the page to be *read-write*, and returns. At each checkpoint, the checkpointer writes out the contents of the changed pages, clears the changed page list, protects the address space to be *read-only* again, and continues. Upon recovery, the state of the last checkpoint is constructed by reading pages from the checkpoint file from most recent to least recent, until every page has been restored.

As stated in Section 2.3 above, our algorithm takes advantage of the observation that if only a few words of a page are changed between checkpoints, then it is wasteful to save the entire page. Instead, our algorithm compresses the page so that its size is on the order of the number of changed words. The algorithm works as follows:

At the beginning of the program, the checkpointer allocates a fixed-size buffer of S_{buffer} bytes. This buffer is not included in any checkpoint of the program. The algorithm continues as in incremental checkpointing: after each checkpoint, the address space is protected to be *read-only*. When a page fault is caught by the checkpointer, the page is put into the changed page list as before, but it is also copied to the checkpointing buffer. Then the page is protected *read-write*, and control returns to the program.

When it is time to take the next checkpoint, there are two copies of each changed page: the current version of the page and the version of the page at the previous checkpoint, held in the checkpointing buffer. There are two ways that we can store the page in the current checkpoint. The first is as before — store the values of the page itself. The second is to store the difference between the current values of the page and the values at the previous checkpoint. This requires taking a bitwise *exclusive or* ($\oplus$) of the current copy of the page and the copy in the checkpointing buffer.

To be formal, let $page_i$ be the current copy of page i, and buf_i be the copy of page i in the checkpointing buffer. Then let $diff_i$ be the bitwise *exclusive or* of $page_i$ and buf_i:

$$diff_i = page_i \oplus buf_i.$$

As stated above, we have the choice of storing $page_i$ or $diff_i$ in the current checkpoint. In our algorithm, we store $diff_i$ because it is easy to compress: any word not changed in $page_i$ is zero in $diff_i$. Therefore, if only a few words have changed in $page_i$, then $diff_i$ will contain a large number of zeros. This property lets us compress $diff_i$ efficiently to a size on the order of the number of changed words in $page_i$.

The compression is based on a bitmap. Let w be the word size of the machine and let p be the page size, both expressed in bytes. Further, let $comp_i$ be the compressed representation of $diff_i$. In the first $\frac{p}{w}$ bits of $comp_i$, we store a bitmap of the non-zero words in $diff_i$. In other words, the j-th bit of the bitmap is set according to whether the j-th word of $diff_i$ is non-zero. The non-zero words of $diff_i$ are then stored in the words following the bitmap in $comp_i$: The k-th non-zero word in $diff_i$ will be the k-th word in $comp_i$ following the bitmap.

The advantage of this compression scheme is its simplicity and speed: $comp_i$ is very efficient to compute, requiring one to three machine instructions per word of $diff_i$. Moreover, the degree of compression is directly proportional to the number of changed words in $page_i$:

$$\text{The number of bytes in } comp_i = \left\lceil \frac{p}{8w} \right\rceil + cw,$$

where c is the number of changed words in $page_i$.

There is a chance that $comp_i$ will be larger than $diff_i$. In such cases, the checkpointer should just store $page_i$ as part of the checkpoint. This means that the recovery routine needs to be able to identify compressed differences as opposed to straight pages. Such a determination can be made trivially by having the checkpointer store an extra bit with each page.

For obvious reasons, we call this algorithm "*compressed differences*."

A major concern of this algorithm is what happens when the checkpoint buffer becomes full. There are three choices that the checkpointer has upon such an event:

1. Force a new checkpoint to be taken. This is the approach taken by Plank and Li's diskless checkpointing algorithm [13]. We consider this unreasonable because it forces the frequency of checkpointing to be a function of the buffer size and program locality, instead of being a variable under the control of the

user. If the buffer is too small, too many checkpoints must be taken, degrading the performance of checkpointing drastically.

2. Revert to incremental checkpointing for the remainder of the checkpointing interval. When the buffer is full, faulting page numbers are simply put into the changed page list, and are not copied to the buffer. At the next checkpoint, only pages in the checkpoint buffer are compressed. We will consider this the standard way of dealing with a full checkpointing buffer, and call this version of the algorithm "*standard compressed differences*."

3. When the checkpointing buffer becomes full, attempt to reclaim some buffer space by compressing pages in the buffer. Specifically, when the buffer is full and a page fault is caught, choose a page buf_i in the checkpointing buffer, calculate $comp_i$, reset the protection of $page_i$ to *read-only* and discard buf_i. Continue doing this until enough compression has occurred that there is a free page in the checkpoint buffer and then continue normally. If a page so compressed generates another page fault (called a *secondary page fault*), then recalculate buf_i, jettison $comp_i$ and mark the page so that buf_i does not get compressed again in an attempt to reclaim buffer space. If there is no room for buf_i in the buffer, then simply jettison $comp_i$ and mark the page so that $page_i$ gets saved in the next checkpoint. In other words, throw the page out of the buffer and have it default to standard incremental checkpointing. When no more buffer space can be reclaimed in this fashion, faulting pages are marked as above for simple incremental checkpointing.

   When it is time to checkpoint, there are three types of changed pages: those that have not been copied to the buffer (including those that have been copied, compressed, and then jettisoned), those that have been copied to the buffer (including those that have been copied, compressed, and then uncompressed) and those that have been compressed. For the first type of page, simply save $page_i$. For the second type, calculate and save $comp_i$. For the third, simply save $comp_i$ as it is stored in the buffer.

   We call this method "*online compressed differences*," as the checkpointer tries to reclaim buffer space by compressing online. The strength of online compression is that we can fit more pages into the checkpointing buffer. However, this does not come for free. By jettisoning compressed pages, we introduce fragmentation into the buffer. Moreover, when a secondary page fault occurs, then the time to copy and compress that page is wasted.

4 Analysis of the Algorithm

In this section, we analyze the theoretical performance of this algorithm in comparison to sequential and incremental checkpointing.

We concentrate on *checkpoint overhead*, which is the amount of time added to the running time of the program as a result of checkpointing. In the equations that follow, there will be four types of variables:

- O_x: This represents the checkpoint overhead of algorithm x.

- S_x: This is the size of x in megabytes (MB).
- R_x: This is the rate of x in MB per second. Although some rates (like page faulting) are more logically presented in pages per second, we assume all rates to be normalized to MB per second.
- F_{comp}: This is the factor of compression, defined as: $\frac{\text{Uncompressed size} - \text{Compressed size}}{\text{Uncompressed size}}$. F_{comp} may only have values between zero (no compression) and one (complete compression).

All variables are assumed to be averages per checkpoint.

Sequential Checkpointing: For standard sequential checkpointing, the overhead is equal to the time it takes to write the entire address space to disk:

$$O_{seq} = \frac{S_{addr.sp}}{R_{disk}} \tag{1}$$

Incremental Checkpointing: The overhead of incremental checkpointing consists of the time to write the changed pages to disk combined with the time to protect the address space and process page faults:

$$O_{inc} = \frac{S_{ch.pgs}}{R_{disk}} + \frac{S_{addr.sp}}{R_{prot}} + \frac{S_{ch.pgs}}{R_{fault}} \tag{2}$$

Incremental checkpointing lowers the overhead of sequential checkpointing whenever the number of changed pages is small enough to offset the time to protect the address space and process page faults. In terms of equations 1 and 2, this is whenever $O_{seq} - O_{inc} > 0$:

$$O_{seq} - O_{inc} = \left(\frac{S_{addr.sp} - S_{ch.pgs}}{R_{disk}}\right) - \left(\frac{S_{addr.sp}}{R_{prot}} + \frac{S_{ch.pgs}}{R_{fault}}\right) \tag{3}$$

Since R_{prot} and R_{fault} are much faster than R_{disk}, and $S_{ch.pgs}$ is usually a fraction of $S_{addr.sp}$, incremental checkpointing is almost invariably an effective technique for lowering the overhead of checkpointing [3, 9, 16].

Standard Compressed Differences: The overhead of standard compressed differences consists of the following components: the time to write the uncompressed pages to disk, the time to write the compressed pages to disk, the time to copy and compress pages in the buffer and the time to perform page handling. If we assume that $S_{ch.pgs}$ is larger than S_{buffer}, then this overhead is:

$$O_{scd} = \frac{S_{ch.pgs} - S_{buffer}}{R_{disk}} + \frac{S_{buffer}(1 - F_{comp})}{R_{disk}} + \frac{S_{buffer}}{R_{c\&c}} + \frac{S_{addr.sp}}{R_{prot}} + \frac{S_{ch.pgs}}{R_{fault}} \tag{4}$$

For simplicity, $R_{c\&c}$ combines the speed of copying pages to the buffer and compressing them.

Thus, standard compressed differences should improve the overhead of incremental checkpointing whenever the savings in checkpoint size offset the added complexity of copying pages to the buffer and compressing them. In terms of equations 2 and 4, this is whenever $O_{inc} - O_{scd} > 0$. Surprisingly, this reduces to a very simple equation:

$$O_{inc} - O_{scd} = S_{buffer}\left(\frac{F_{comp}}{R_{disk}} - \frac{1}{R_{c\&c}}\right) \tag{5}$$

Thus, we expect standard compressed differences to improve the overhead of incremental checkpointing whenever the compression factor is greater than the ratio of disk speed to copy and compression speed: $F_{comp} > \frac{R_{disk}}{R_{c\&c}}$. The extent of improvement depends on the magnitude of F_{comp} and S_{buffer}.

Online Compressed Differences: The overhead of online compressed differences is more complex. To quantify it, we split $S_{ch.pgs}$ into five components:

- S_{comp}: MB in pages that are compressed in the buffer at checkpoint time.
- $S_{uncomp1}$: MB in pages that are uncompressed in the buffer at checkpoint time, and have not been compressed yet.
- $S_{uncomp2}$: MB in pages that are uncompressed in the buffer at checkpoint time, but were compressed at one point in an attempt to gain more buffer space.
- S_{dirty1}: MB in dirty pages that have not been copied to the buffer.
- S_{dirty2}: MB in dirty pages that were copied to the buffer, compressed, and then jettisoned from the buffer.

It should be clear that $S_{comp} + S_{uncomp1} + S_{uncomp2} + S_{dirty1} + S_{dirty2} = S_{ch.pgs}$. We define the "effective buffer size" to be sum of all bytes that get compressed in the checkpoint file: $S_{ebuffer} = S_{comp} + S_{uncomp1} + S_{uncomp2}$. The overhead of online compressed differences can then be quantified as follows:

$$O_{ocd} = \frac{(1 - F_{comp})S_{ebuffer} + S_{dirty1} + S_{dirty2}}{R_{disk}} + \frac{S_{ebuffer} + 2S_{uncomp2} + S_{dirty2}}{R_{c\&c}} + \frac{S_{addr.sp} + S_{comp} + S_{uncomp2} + S_{dirty2}}{R_{prot}} + \frac{S_{ch.pgs} + S_{uncomp2} + S_{dirty2}}{R_{fault}} \quad (6)$$

This is not a simple equation. However, it encapsulates the crux of online compressed differences, which is the effective buffer size. If the effective buffer size is much larger than the buffer size (i.e. $S_{ebuffer} \gg S_{buffer}$), then online compressed differences should improve overhead: more bytes will be compressed, and the savings in writing fewer bytes to disk will compensate for extra copying, compression and page fault handling. However, if the effective buffer size is not significantly larger than the buffer size, then the extra work of copying, compression and page fault handling will increase overhead. Finally, it is possible for the effective buffer size to be smaller than the buffer size. This is because the variable size of compressed pages can lead to fragmentation in the buffer. If this is the case, then online compressed differences will obviously perform worse than standard compressed differences.

5 Implementation and Experiments

For our implementation, we modified **libckpt**, a general-purpose checkpointer for Unix-based uniprocessors [16]. **Libckpt** implements incremental checkpointing by using the standard Unix system calls for page protection: `mprotect()` and `signal()`. We modified this implementation to include both standard and online compressed differences.

We checkpointed several long-running programs on a Sun Sparcstation 2 running SunOS 4.1.3. This machine has 16 MB of physical memory and a page size of 4096 bytes. The rates of page fault handling and compression are as follows: $R_{prot} = 6,892$ MB/sec, $R_{fault} = 8.679$ MB/sec, and $R_{c\&c} = 1.138$ MB/sec. Checkpoints are written to a Hewlett Packard HP6000 disk via NFS at a rate of 0.1797 MB/sec.

Thus from equation 3 we predict that incremental checkpointing will improve the overhead of checkpointing whenever $\left(\frac{S_{addr.sp}}{S_{ch.pgs}}\right) > 1.02$. Moreover, from equation 5, standard compressed differences should improve the overhead of incremental checkpointing whenever $F_{comp} > 0.16$.

We tested the performance of checkpointing on a variety of long-running programs: **LOG** is a simulation program that computes the messages to be logged according to a critical path based logging algorithm [24]. **CELL** is a program that executes a 2048×2048 grid of cellular automata for fifteen generations. **CONTOUR** calculates altitude contours on a 2816×2179 byte-map. **SOLVE** uses the `dgesv` subroutine from LAPACK [25] to solve a linear system with 1000 equations, 1000 unknowns, and 500 right-hand sides. **WATER** is the program `STSWM` from the National Center for Atmospheric Research. The program implements a shallow water model based on the spectral transform method [26]. The instance used here is "Zonal Flow over a Mountain" from their test suite, modeled at 15-minute intervals for six hours. **MCNF** solves the multicommodity network flow problem using the simplex method [27]. The instance used here runs on a network of 100 vertices and 50 commodities.

5.1 Basic Data

Table 1 shows the basic data of each program and the results of sequential and incremental checkpointing with a checkpoint interval of two minutes. As in the analysis above, the values of the variables are given in average time or space per checkpoint. All values shown in this and subsequent tables are averages of six or more program runs. All times are wall clock times[1].

Application	Running Time (sec)	Running Time (mm:ss)	# of Ckps	$S_{addr.sp}$ (MB/ckp)	$S_{ch.pgs}$ (MB/ckp)	$\frac{S_{addr.sp}}{S_{ch.pgs}}$	O_{seq} (sec/ckp)	O_{inc} (sec/ckp)	% Improvement
LOG	1345	(22:25)	11	1.592	1.551	1.026	9.04	9.82	-8.63
CELL	1194	(19:54)	9	8.069	7.679	1.051	44.62	48.78	-9.32
CONTOUR	766	(12:46)	6	11.755	6.877	1.709	70.74	41.97	40.67
SOLVE	775	(12:55)	6	11.545	6.167	1.872	68.52	37.35	45.49
WATER	1458	(24:18)	12	13.574	4.333	3.133	77.80	33.27	57.24
MCNF	1109	(18:29)	9	23.187	1.296	17.891	146.65	7.46	94.91

Table 1: Basic Data for the Application Programs

From Table 1 we see that in most cases, incremental checkpointing is a large improvement over sequential checkpointing, decreasing the per-checkpoint overhead by up to 95%. This corroborates previous results concerning incremental checkpointing [3, 9, 16].

5.2 Standard Compressed Differences

[1] The raw data is available by anonymous `ftp` to `cs.utk.edu`. See the file `pub/plank/scd/README`. The raw data is also available on the web at `http://www.cs.utk.edu/~plank/plank/scd/scd.html`.

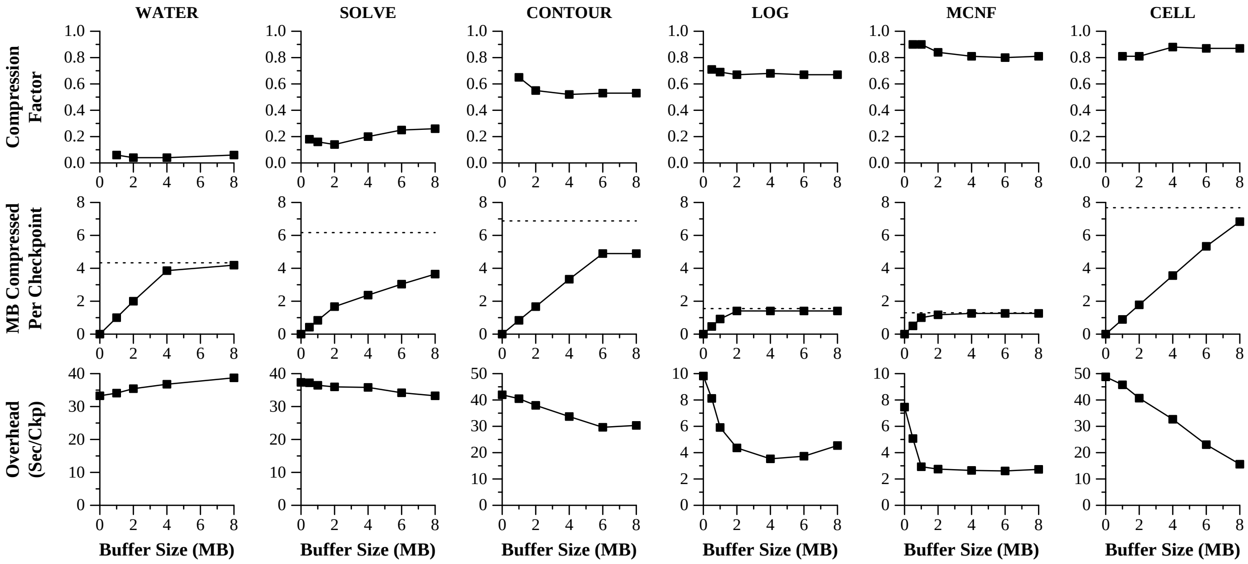

Figure 1: Relevant data for standard compressed differences

To test the effectiveness of standard compressed differences, we checkpointed at the same intervals using buffer sizes from 512 KB to 8 MB. The relevant data from these experiments is contained in Figure 1.

For each application, three graphs are plotted as a function of the buffer size: the compression factor (F_{comp}), the number of megabytes compressed per checkpoint, and the per-checkpoint overhead of checkpointing. Note that we include data for a buffer size of zero — this is simply the data for incremental checkpointing.

The applications are ordered by their compressibility. Compression factors range from very low (e.g. 0.04 for **WATER**) to very high (0.90 for **MCNF** and **CELL**), and are fairly unaffected by the buffer size. As predicted by Equation 5, a compression factor of 0.16 denotes the break-even point for the algorithm — when there is less compression, as in the **WATER** application, the overhead of checkpointing increases relative to incremental checkpointing; when more compression occurs, then the performance of checkpointing improves.

The amount of improvement is dictated by the compression factor and the number of bytes compressed. The middle graph for each application shows the megabytes compressed per checkpoint. This number increases as the buffer size increases until the buffer size becomes larger than the incremental checkpoint size (shown by the dotted line), at which point it remains constant.[2]

[2] In some applications (e.g. **SOLVE** and **CONTOUR**) the number of bytes compressed levels off at a number smaller than the average incremental checkpoint size. This is because of newly `malloc()`'d data. Such data starts in an undetermined state under SunOS, and is not considered a candidate for compression. However, it must be included in the checkpoints.

Compression Factors

At this point, we discuss the reasons for the widely varying compression factors. As discussed in section 3, we expect compression to succeed when only a fraction of words in a page are altered between checkpoints. In the **WATER** and **SOLVE** programs, this is not the case. In **WATER**, every value on a dense grid of variables is updated every iteration, and there are multiple iterations per checkpoint. Thus, all pages containing the grid are completely updated between checkpoints, and little compression is possible. In **SOLVE**, successive columns of a matrix A are updated in each iteration. A is ordered so that adjacent values in a column are adjacent in memory, meaning that pages are updated in their entirety. Like **WATER**, this leads to small compression factors.

Compression factors are much higher in the other four programs because the memory updates are much sparser per page. In **CONTOUR**, two 2-dimensional byte-maps are held in memory — the input map, and the output map. There is one iteration for each altitude, and the result of that iteration is that each location on the input map with the proper altitude is set on the output map. This leads to sparse page updates and thus results in good compression. In both **LOG** and **MCNF**, the calculations are graph-based, involving a great deal of pointer-chasing. Thus, the memory access patterns are relatively sparse per page, and also result in good compression. Finally, the **CELL** program maintains a grid of values, and for each iteration updates each value as a function of that value and its neighbors. For certain inputs, it is often the case that many values remain unchanged from iteration to iteration. This once again leads to excellent compression.

5.3 Online Compressed Differences

To test the effectiveness of online compressed differences, we checkpointed at the same intervals using the same buffer sizes as before. The relevant data is contained in Figures 2 and 3. The **WATER** application is omitted because the poor compression factor of this application renders all algorithms based on compressed differences useless.

The main conclusion to draw from this data is that online compressed differences rarely improve the overhead of checkpointing over standard compressed differences. In fact, the only time that they showed improvement in this experiment was in the **CELL** application when a 6MB buffer was used. In all applications, there were cases where online compressed differences degraded the performance of checkpointing. This is due to two reasons:

1. The effective buffer size (represented by the striped regions in the bar graphs) was not bigger than the actual buffer size. Because of secondary page faults and fragmentation within the checkpointing buffer, online compressed differences rarely allowed more bytes to be compressed than standard compressed differences. The only cases when the effective buffer size was bigger than the actual buffers (**SOLVE**-8MB, **MCNF**-2MB and **CELL**-6MB) were when the buffer size was almost big enough to hold all of the changed pages. Thus, while standard compressed differences can always improve the overhead of incremental checkpointing regardless of buffer size, online compressed differences do not always improve

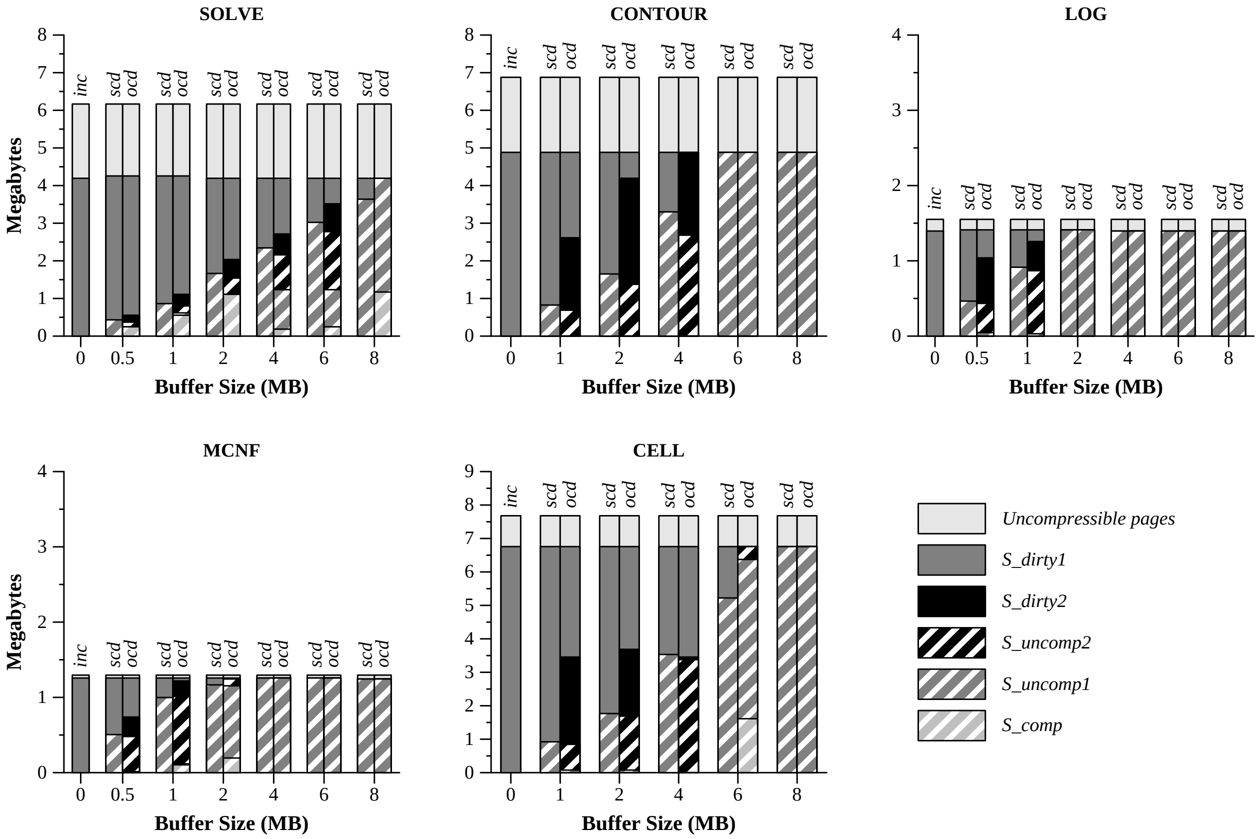

Figure 2: Size information for online compressed differences

the overhead of standard compressed differences.

2. The overhead of secondary page faults was too great. In the bar graphs of Figure 3, the black regions (both striped and solid) represent the pages that caused secondary page faults. Where these regions are significantly large (e.g. **SOLVE**-6MB and **CONTOUR**-4MB), the overhead of checkpointing was correspondly worse than with standard compressed differences.

6 Conclusions

We have described, analyzed, and experimentally explored the effectiveness of two improvements to incremental checkpointing called standard and online compressed differences. The crux of these algorithms is to use fast compression combined with buffering to reduce the size of incremental checkpoints, thereby decreasing the overhead.

Our analysis has determined that standard compressed differences improve the overhead of incremental checkpointing whenever the compression factor is greater than the ratio of disk speed to copy/compression speed: $F_{comp} > \frac{R_{disk}}{R_{c\&c}}$. In our experiments, this ratio is 0.16, and was exceeded in five of the six applications.

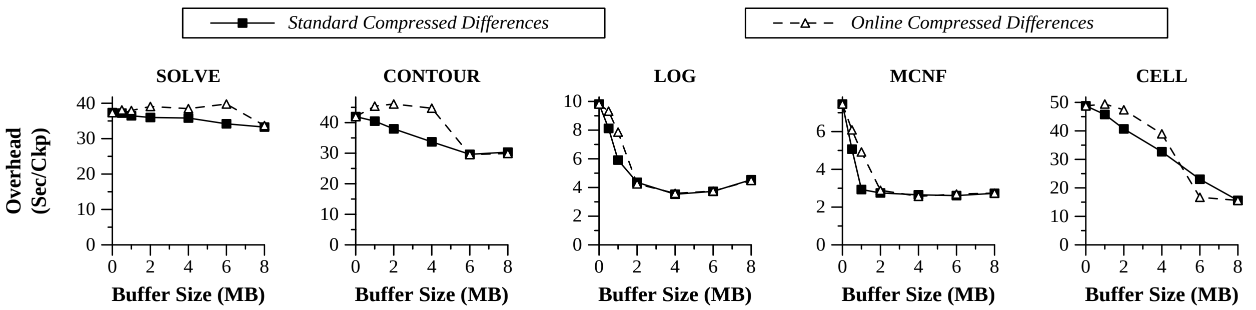

Figure 3: Overhead of online compressed differences

Correspondly, those five applications improved the overhead of incremental checkpointing. The amount of improvement depended on both the compression factor and the size of the buffer, larger buffers showing more improvement than smaller ones. The important thing to note is that regardless of the size of the checkpointing buffer, as long as the compression factor was greater than 0.16, standard compressed differences improved the overhead of checkpointing.

The compression factor is related to the memory access patterns of the program. Since the compression essentially shrinks each page to number of changed words between checkpoints, pages that are sparsely written compress much better than those that are densely written. Thus, programs that update every element of a dense grid or matrix (e.g. **WATER** and **SOLVE**) compress far worse than those that update their data space more sparsely or randomly.

Online compressed differences compress pages before it is time to checkpoint in an attempt to manufacture more buffer space. This runs the risk of having to uncompress the pages if they are prematurely accessed, which may lead to fragmentation in the buffer as well. Experiments show that unless the checkpointing buffer is close to the size of the next incremental checkpoint, online compressed differences actually increase the overhead of checkpointing. This is due to fragmentation in the buffer combined with the extra cost of handling secondary page faults.

The experiments presented in this paper were all performed on a uniprocessor. We expect standard compressed differences to improve the overhead of checkpointing in parallel and distributed environments, especially those that use a centralized stable storage, because of the high ratio of computational power to disk speed.

References

[1] K. M. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Transactions on Computers*, vol. 21, pp. 546–556, June 1972.

[2] N. S. Bowen and D. K. Pradhan, "Processor and memory-based checkpoint and rollback recovery," *IEEE Computer*, vol. 26, no. 2, pp. 22–31, 1993.

[3] S. I. Feldman and C. B. Brown, "Igor: A system for program debugging via reversible execution," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, vol. 24, pp. 112–123, Jan 1989.

[4] L. D. Wittie, "Debugging distributed C programs by real time replay," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, vol. 24, pp. 57–67, Jan 1989.

[5] D. R. Jefferson, "Virtual time," *ACM Trans. on Programming Languages and Systems,*, vol. 7, pp. 404–425, July 1985.

[6] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the Unix kernel," in *Conference Proceedings, Usenix Winter 1992 Technical Conference*, (San Francisco, CA), pp. 283–290, January 1992.

[7] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and its applications," in *25th International Symposium on Fault-Tolerant Computing*, (Pasadena, CA), pp. 22–31, June 1995.

[8] P. R. Wilson and T. G. Moher, "Demonic memory for process histories," in *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 330–343, June 1989.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *11th Symposium on Reliable Distributed Systems*, pp. 39–47, October 1992.

[10] C.-C. J. Li and W. K. Fuchs, "CATCH – Compiler-assisted techniques for checkpointing," in *20th International Symposium on Fault Tolerant Computing*, pp. 74–81, 1990.

[11] J. S. Plank and K. Li, "Ickp — a consistent checkpointer for multicomputers," *IEEE Parallel & Distributed Technology*, vol. 2, pp. 62–67, Summer 1994.

[12] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 874–879, August 1994.

[13] J. S. Plank and K. Li, "Faster checkpointing with $N+1$ parity," in *24th International Symposium on Fault-Tolerant Computing*, (Austin, TX), pp. 288–297, June 1994.

[14] E. N. Elnozahy, *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Rice University, 1993.

[15] J. S. Plank, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, January 1993.

[16] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "**Libckpt**: Transparent checkpointing under unix," in *Conference Proceedings, Usenix Winter 1995 Technical Conference*, pp. 213–223, January 1995.

[17] A. Appel and K. Li, "Virtual memory primitives for user programs," in *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA), pp. 96–107, April 1991.

[18] R. H. B. Netzer and M. H. Weaver, "Optimal tracing and incremental reexecution for debugging long-running programs," in *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, (Orlando, FL), pp. 313–325, June 1994.

[19] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys*, vol. 21, pp. 557–589, December 1989.

[20] V. Cate and T. Gross, "Combining the concepts of compression and caching for a two-level filesystem," in *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA), pp. 200–211, April 1991.

[21] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line data compression in a log-structured file system," in *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–9, ACM, October 1992.

[22] F. Douglis, "The compression cache: Using on-line compression to extend physical memory," in *Conference Proceedings, Usenix Winter 1993 Technical Conference*, (San Diego, CA), pp. 519–529, January 1993.

[23] T. A. Welch, "A technique for high-performance data compression," *IEEE Computer*, vol. 17, pp. 8–19, June 1984.

[24] R. H. B. Netzer, S. Subramanian, and J. Xu, "Critical-path-based message logging for incremental replay of message-passing programs," in *14th International Conference on Distributed Computing Systems*, (Poznan, Poland), June 1994.

[25] E. Anderson *et al*, *LAPACK User's Guide – Release 2.0.* Philadelphia, PA: SIAM, 1994.

[26] J. J. Hack, R. Jakob, and D. L. Williamson, "Solutions to the shallow water test set using the spectral transform method," Tech. Rep. TN-388-STR, National Center for Atmospheric Research, Boulder, CO, 1993.

[27] J. Kennington, "A primal partitioning code for solving multicommodity flow problems (version 1)," Tech. Rep. IEOR-79009, Southern Methodist University, 1979.