5-1-2004

A Holesome File System

Darren Erik Vengroff
Brown University

David Kotz
Dartmouth College

A Holesome File System

Darren Erik Vengroff
Department of Computer Science
Brown University and Duke University

David Kotz
Dartmouth College
dfk@cs.dartmouth.edu

Dartmouth College Computer Science
Technical Report TR2004-497

Written July 17, 1995
Released May 2004

Abstract

We present a novel approach to fully dynamic management of physical disk blocks in Unix file systems. By adding a single system call, `zero`, to an existing file system, we permit applications to create *holes*, that is, regions of files to which no physical disk blocks are allocated, far more flexibly than previously possible. `zero` can create holes in the middle of existing files.

Using `zero`, it is possible to efficiently implement applications including a variety of databases and I/O-efficient computation systems on top of the Unix file system. `zero` can also be used to implement an efficient file-system-based paging mechanism. In some I/O-efficient computations, the availability of `zero` effectively doubles disk capacity by allowing blocks of temporary files to be reallocated to new files as they are read.

Experiments on a Linux *ext2* file system augmented by `zero` demonstrate that where their functionality overlaps, `zero` is more efficient than `ftruncate()`. Additional experiments reveal that in exchange for added effective disk capacity, I/O-efficient code pays only a small performance penalty.

Note: After writing this paper we learned of an earlier paper, which mentions "The ZERO procedure to punch holes in a file" on page 139.

> Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. "NFS Version 3: Design and Implementation," *Proceedings of the 1994 Summer USENIX Conference*, pages 137–152. June 1994.

Nonetheless, our paper outlines some applications of such a facility and evaluates the performance benefits to those applications.

A Holesome File System

Darren Erik Vengroff[*]
Department of Computer Science
Brown University & Duke University
dev@cs.duke.edu

David Kotz[†]
Department of Computer Science
Dartmouth College
dfk@cs.dartmouth.edu

Abstract

We present a novel approach to fully dynamic management of physical disk blocks in Unix file systems. By adding a single system call, `zero()`, to an existing file system, we permit applications to create *holes*, that is, regions of files to which no physical disk blocks are allocated, far more flexibly than previously possible.

Using `zero()`, it is possible to efficiently implement applications including a variety of databases and I/O-efficient computation systems on top of the Unix file system. `zero()` can also be used to implement an efficient file-system-based paging mechanism. In some I/O-efficient computations, the availability of `zero()` effectively doubles disk capacity by allowing blocks of temporary files to be reallocated to new files as they are read.

Experiments on a Linux *ext2* file system augmented by `zero()` demonstrate that where their functionality overlaps, `zero()` is more efficient than `ftruncate()`. Additional experiments reveal that in exchange for added effective disk capacity, I/O-efficient code pays only a small performance penalty.

1 Introduction

One of the primary functions of any operating system is to manage resources on behalf of applications. This is particularly true of bulk resources, such as memory and storage space, that may be required in different amounts at different times during the execution of an application. The `malloc()` and `free()` functions provided in the standard C library on Unix systems are a familiar example of bulk resource management.

Unfortunately, Unix does not provide a fully capable management mechanism for physical disk space. Allocation of physical disk space is done implicitly, as necessary, by the `write()` system call, but fine-grained deallocation is not well supported. This technique has drawbacks for a number of important application classes, including database-management systems and applications of I/O-efficient computation [Ven95, Ven94]. These applications manage large quantities of disk-resident data, so efficient use of physical disk space, including the ability to free unneeded blocks, is critical to their performance. If physical disk space cannot be easily freed, then the tendency is for significant numbers of unneeded "garbage" blocks to remain on disk. The result is that not all of the disk can be put to useful work, which artificially limits the amount of data that can be manipulated. When restricted to rely on standard Unix file-system semantics, many I/O-efficient computations are forced to dedicate up to half of their disk space to garbage.[1]

Because Unix file systems do not meet the needs of applications that demand the ability to manage physical disk blocks, such applications typically resort to working with raw devices. Although this permits better control over the allocation of physical disk space, it prevents the application from taking advantage of any other potentially useful services provided by the Unix file system.

In this paper, we demonstrate that by adding a single system call, `zero()`, to Unix, we can provide fully dynamic management of physical disk space. In some cases, applications can make direct use of `zero()`,

[*] Supported in part by the U.S. Army Research Office under grant DAAH04–93–G–0076 and by the National Science Foundation under grant DMR–9217290. Portions of this work were conducted while visiting the University of Michigan, the University of Washington, and Syracuse University.

[†] Supported in part by Dartmouth College, by NSF under grant number CCR 9404919, by NASA Ames Research Center under Agreement Number NCC 2-849, and by Syracuse University.

[1] We discuss the reasons in Section 4.1.

though more typically they will rely on an intermediate programming interface, such as TPIE [Ven95, Ven94], or a block-management library, such as `libba`, which we describe in Section 4.2. Furthermore, we show how some virtual-memory systems could benefit from the availability of `zero()` by allowing swap files to grow and shrink as necessary. Finally, we describe experiments in Sections 5 and 6 that illustrate the efficiency of an implementation of `zero()` inside the Linux *ext2* file system.

2 Existing Unix File Systems

A wide variety of computers, including personal computers, workstations, and supercomputers (both parallel and vector) use some variant of the Unix operating system [RT78, Tho78]. The Unix file system treats files as addressable, growable sequences of bytes. To support this abstraction, disk space is divided into fixed-size blocks, and the metadata for each file includes a set of pointers to disk blocks. The file's *inode* contains 12 pointers to *direct blocks*, a pointer to an *indirect block*, a pointer to a *doubly indirect block*, and a pointer to a *triply indirect block*. Indirect blocks contain more pointers to data blocks, doubly indirect blocks contain more pointers to indirect blocks, and triply indirect blocks contain more pointers to doubly indirect blocks. Pointers are implemented as block numbers of corresponding to the physical location of the block on the disk. For more details see [Bac86, LMKQ89, MJLF84].

In Unix, when an application seeks past the end of an open file and then issues a write request, the file is extended to accommodate the new data. That is, the file size is updated to include the new data. Reading file positions between the old end of file and the beginning of the new data produces zeroes. Most versions of Unix implement large zero regions created in this way as *holes* in the file. Within a hole, no physical blocks are allocated to hold the zero data; instead, the corresponding block pointers in the inode metadata are simply set to zero [LMKQ89, page 194]. In this way, extremely large but sparse files can be represented with relatively little storage space. This is a significant feature of Unix upon which `zero()` depends.

2.1 The Linux *ext2* File System

The Linux operating system [Joh95, Wir95] is a complete re-implementation of Unix that is distributed free of charge under the terms of the Free Software Federation's GNU General Public License [Fed]. Linux runs on platforms based on the Intel 80x86 family of CPUs. From the user's perspective, Linux is remarkably like Unix, and can be considered equivalent for the purposes of this paper. The primary file-system type supported by Linux is the *ext2* file system, designed by Rémy Card. The inode metadata structure is identical to that described above.

3 The `zero()` System Call

Our `zero()` system call is designed to allow for the creation of holes within files. Its name, and additional aspects of its behavior, are derived from the fact that reading from a hole returns data consisting of all zeroes.

The format of the `zero()` system call is

```
int zero(int fd, unsigned int nbytes);
```

Its arguments are

`fd`

A file descriptor corresponding to an open file.

`nbytes`

The number of bytes that should be zeroed, starting at the current file offset.

From the perspective of the application, the effect of `zero()` is to replace `nbytes` bytes of data in the file, beginning at the current logical offset, with zeroes. It is as if `write()` had been called with the same `fd` and `nbytes` and with a buffer of `nbytes` zeroes. Unlike `write()`, however, `zero()` frees any blocks that

are completely zeroed, creating a hole in the file, and only writes zeroes over partially affected blocks. Our implementation also frees any indirect blocks (single, double, or triple) that contain only zero pointers. The call is particularly efficient if the file pointer is aligned on a block boundary and the number of bytes is a multiple of the block size, since in such cases only metadata blocks are updated.

If a call to `zero()` succeeds, the return value is the number of bytes that were zeroed. If an error occurs, then the return value is the negated error code. Zeroing past the current EOF extends the EOF appropriately. If `nbytes` is positive, then both the `mtime` and `ctime` of the file are updated.

4 Applications of `zero()`

4.1 I/O-Efficient Computation

In recent years, I/O-efficient algorithms for a wide variety of problems have been developed. These include algorithms from the domains of sorting [AV88, Arg94, NV90, NV93, VS94], permuting [CSW94, Cor93, Cor92], computational geometry [GTVV93], line segment intersection [AVV95], graph algorithms [CGG+95], and scientific computation [VV95]. The TPIE library [Ven95, Ven94] supports efficient implementation of many of these algorithms.

A common theme in many of these algorithms is that they make a series of passes through their data; each pass reads most or all of the data from the previous pass and writes a similar amount of data, which becomes the input for the next pass. Because the reads and writes are not necessarily sequential, ordinary Unix pipes cannot generally be used between passes. Instead, these algorithms are typically implemented so that each pass writes a temporary file. Only when the $i+1$st pass has completed is it safe to delete the temporary file `tfi` that was its input (and the output of the ith pass). Thus, at that time both `tfi` and `tf(i+1)` are stored on disk. If both are the same size, then neither can be larger than half of the disk space available before the program ran. If `zero()` is available, however, then physical blocks of `tfi` can be released as they are read into main memory. These blocks can then be re-used by `tf(i+1)` as it is written. The net result is that the size of problem that can be solved with a given amount of disk space is effectively doubled, because each temporary file can occupy all available space.

4.2 The `libba` Block Allocation Library

Not all applications that benefit from the ability to manage physical disk space are amenable to implementation in the context of TPIE. In particular, applications that use external-memory dynamic data structures, such B-trees [Com79], grid files [NHS84], or R-trees [Gut84] and their variants [BKSS90, SRF87], need to be able to allocate and deallocate space a block at a time. These data structures are widely used in a variety of database systems, thus there is ample motivation to support them.

Without `zero()`, it is possible to implement applications that allocate and deallocate fixed sized disk blocks within a Unix file system, but there are drawbacks. The standard technique is to maintain a free list that points to unneeded blocks within a file, so that when a request to allocate a block is received, one of the free blocks is used. Only when the free list is empty will another block be added to the end of the file. Deallocation requests are handled by putting the block to be deallocated onto the free list. The problem with this technique is that because deallocated blocks are not physically released back to the file system, the file's physical space never decrease even when the application needs little space.

With `zero()`, the same approach can be used, but deallocated blocks can also be physically removed from the file. Thus, at any given time, the physical size of the file is only as large as necessary to meet the current demands of the application.[2]

Functions to allocate and deallocate blocks can be implemented as a user-level library called `libba` on top of the `zero()` system call. `libba` provides six functions:

`bfd ba_create(int fd)`

Create a block allocation on top of an existing file, whose descriptor is given.

[2] This is not entirely true, but it is close. If the logical size of the file is extremely large but only a small number of physical data blocks are allocated then some additional physical blocks are likely to be needed to store indirect blocks.

`int ba_size(bfd bfile)`

Return the size of a block in the block file. This size depends on the implementation and/or the underlying device.

`bid ba_alloc(bfd bfile)`

Allocate a block in the file and return its block identifier.

`int ba_free(bfd bfile, bid block)`

Free a block in the given block file with the given id.

`int ba_read(bfd bfile, bid block, void *buf)`

Read the contents of the identified block into the main memory buffer pointed to by `buf`.

`int ba_write(bfd bfile, bid block, void *buf)`

Write the contents the buffer pointed to by `buf` to the identified block.

Database systems are not the only application that can benefit from the use of `libba`. The library can also be used as the basis for a file-system virtual-memory pager. Some variants of Unix support swapping to files instead of dedicated swap partitions [TRY+87, Wir95]. Normally, swapping is done either to a file of fixed physical size, as in Linux [Wir95], or to a file whose physical size is as large as the maximum amount of virtual memory in use at any time since the system was booted, as in some versions of the Mach operating system [TRY+87]. Using a file managed by `libba` as swap space allows more flexibility is system administration, since no fixed upper limit on the swap file size has to be established and only as much physical disk space as is needed is used at any given time.

5 Experimental Methodology

For the sake of experimentation, `zero()` was implemented as an extension to the *ext2* file system in the LINUX operating system, kernel version 1.1.59. The implementation of `zero()` was modeled after `write()`, except that instead of writing data to buffers in the buffer cache, we release the buffers and replace metadata pointers to them by zero pointers.

To verify that `zero()` performs as expected, we ran two classes of benchmarks. First, we ran microbenchmarks to verify that `zero()` was working as intended, and to compare its performance to the existing `ftruncate()` system call.[3] Second, we ran I/O-efficient computation benchmarks to assess the costs and benefits of using `zero()` for such computations.

5.1 Testbed System

All benchmarks were run on a 100MHz Intel Pentium processor running our modified Linux 1.1.59 kernel. I/O was done through a Buslogic BT-946C fast SCSI PCI host adapter to a Connor 1080S disk. The partition on which the tests were run had approximately 415Mb of free space. All tests were run while the system was otherwise idle.

5.2 Microbenchmarks

To verify that `zero()` behaved as expected, we implemented three small benchmarks to test it in controlled environments. The benchmarks were:

1. write a file of N blocks, then `zero()` all N blocks at once.
2. write a file of N blocks, then `ftruncate()` the file to 0 length.
3. write a large file, then `zero()` N blocks in the middle of the file.

[3] `ftruncate()` provides a subset of the functionality of `zero()` by permitting the de-allocation of physical disk blocks at the ends of files only.

We designed the first two experiments to compare the cost of `zero()` to that of the existing `ftruncate()` system call. As the results in Section 6.1 indicate, `zero()` actually performed better than `ftruncate()` in our tests. The third experiment bears greater resemblance to expected uses of `zero()`. Zeroing N blocks in the middle of a large file was more expensive than zeroing an N-block file, especially for small N, because of the overhead of manipulating indirect and doubly-indirect blocks.

We repeated each of these experiments five times, and report the mean execution time. Because each experiment changed the file system, we measured both the time for the system call, and the time for the system call followed by a call to `fsync()`, which forces all changes to the file's data and metadata to be flushed to disk. The former represents the latency seen by an application, whereas the latter represents the true impact of the call on the system.

5.3 I/O-Efficient Computation Benchmarks

To test the use of `zero()` for I/O-efficient computing, we modified TPIE [Ven95, Ven94], a Transparent Parallel I/O Environment, to use `zero()` to free blocks of temporary files as they are read. TPIE presents a high-level interface to application programs that hides low-level I/O operations, including `zero()`. Thus, we only had to make trivial changes to application programs for them to take advantage of `zero()` through TPIE.

Using our modified version of TPIE, we ran two I/O-efficient computation benchmarks: `nas_ep`, a benchmark chosen from the NAS parallel benchmark suite [BBB+94], whose TPIE implementation is described in [VV95]; and `s24`, a benchmark that merge-sorts records consisting of a 4-byte integer key and 20 additional bytes of data. We ran each benchmark for a variety of problem sizes, up to the limits imposed by available disk space.

6 Experimental Results

6.1 Microbenchmarks

The benchmarks designed to compare the performance of `zero()` and `ftruncate()` produced pleasantly surprising results, as illustrated in Figure 1. `zero()` proved not only the equal of `ftruncate()`, but was actually more efficient, even when followed by `fsync()` to flush all changes to the disk. We believe, from inspecting the code, that the difference was due to a more efficient implementation.

The second microbenchmark also produced interesting results, as shown in Figure 2. Most notably, the time decreased as the number of blocks zeroed increased. We speculate that was due to the increasing number of indirect blocks that could be freed instead of written back to disk.

6.2 I/O-Efficient Computation Benchmarks

The NAS EP benchmark generates pairs of independently distributed, Gaussian random variates. For problem size N, the benchmark generates N pairs of 8-bit floating point numbers, then scans them, producing approximately $N\pi/4$ pairs as output. Without `zero()`, we need enough free disk space to hold $N(1+\pi/4) \approx 1.78N$ pairs. With `zero()`, we only need enough space to hold N pairs. See [BBB+94] for a complete description of the benchmark and [VV95] for a description of its implementation in TPIE.

The *s24* benchmark merge sorts N records consisting of 24 bytes each. Merge sort makes a series of passes through the data, each of which permutes its input data. Because each pass writes exactly as much data as it reads, the total size of the problem that can be solved without `zero()` is limited to half of the available disk space. With `zero()`, problems almost as large as the available disk space can be solved. The result is an essentially in-place external-memory sort.

The performance of these two benchmarks is illustrated in Figures 3 and 4. In both cases, the availability of `zero()` allowed us to run larger instances of the problems than we could otherwise have done. The space savings were offset by a small increase in the execution time.

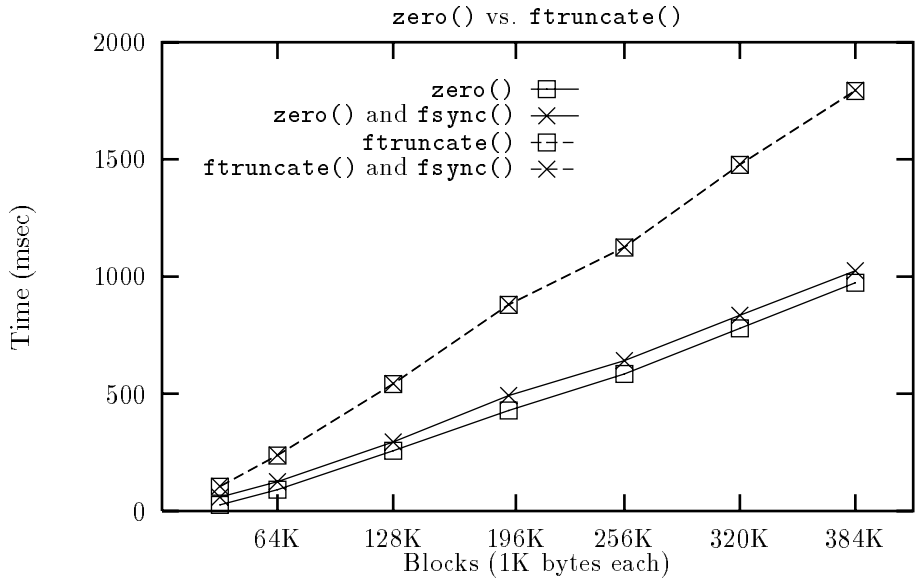

Figure 1: The time to `zero()` an entire file of N blocks, for N from 32K to 384K (i.e., file sizes from 32 Mb to 384 Mb). For comparison, we also show the time to `ftruncate()` a similar file to zero size. In both cases, we plot both the time for the system call alone, and for the system call followed by `fsync()`, which flushes all changes to disk. For `ftruncate()` there is essentially no difference in the times, since Linux's implementation of `ftruncate()` writes its changes back to the disk, leaving nothing for `fsync()` to do.

7 Summary

We present `zero()`, a new system call for Unix file systems. The `zero()` system call allows a programmer to create "holes" in the middle of an existing file, which subsequently reads as zeroes.

The beauty of the `zero()` system call is in its simplicity, and its natural semantic fit with other Unix file-system behavior. The power of the `zero()` system call, however, is in its connection to the Unix implementation of file "holes." Large (block-sized) holes in Unix files are not allocated any physical disk space, which gives `zero()` the side-effect of freeing disk space associated with the newly zeroed region. This feature makes it possible for applications to control their physical disk-space usage more accurately and efficiently than before.

In our implementation we use `zero()` in the TPIE I/O library to support I/O-intensive applications. We show that in I/O-efficient computation applications, the availability of `zero()` allowed us to solve much larger problems (typically close to a factor of two larger) with tolerable increases in execution time.

Thus far, we have only scratched the surface in terms of applications of `zero()`. We plan to continue this research by conducting in-depth studies of the applicability and performance of `zero()` for a variety of additional applications.

Acknowledgments

The authors thank the lab staff at the CASE Center at Syracuse University for assembling the hardware platform on which `zero()` was originally implemented, and particularly Nilesh Patel, for installing Linux.

We thank Jeff Vitter of the Department of Computer Science at Duke University for providing the machine on which the experiments were conducted.

Darren Vengroff also thanks Yale Patt of the ACAL Lab at the University of Michigan, Ed Lazowska of the Department of Computer Science and Engineering at the University of Washington, and their respective staffs for providing logistical support during his visits to their institutions.

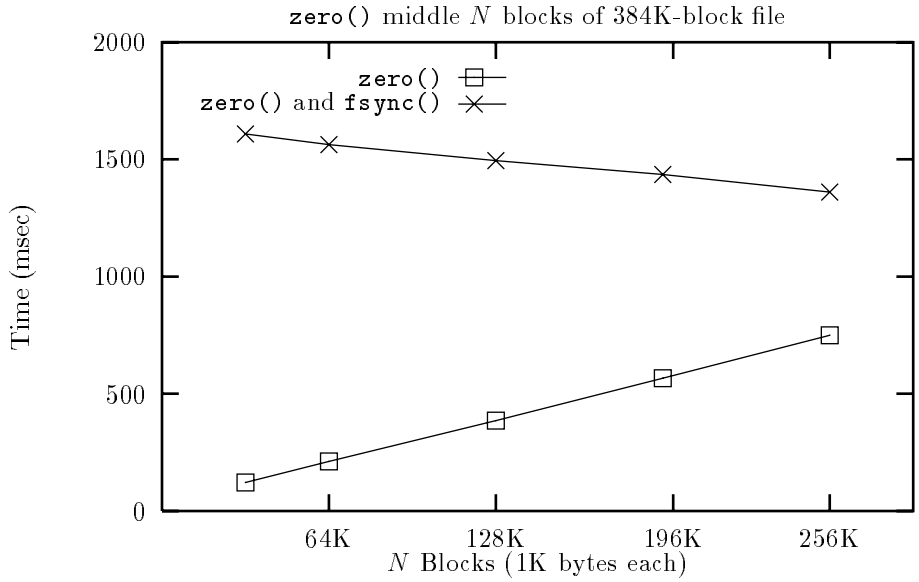

Figure 2: The time to `zero()` N blocks of data from the exact middle of a file of 384K blocks (= 384Mb). We plot both the time for the system call alone, and for the system call followed by `fsync()`, which flushes all changes to disk.

References

[Arg94] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. Technical Report RS-94-16, BRICS, Univ. of Aarhus, Denmark, 1994.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AVV95] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in georgraphic information systems. In *Proc. 3rd Europ. Symp. on Algorithms*, LNCS, Corfu, Greece, September 1995. Springer-Verlag. To appear.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lansinki, R. Schrieber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, March 1994.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD Conference*, pages 322–331, 1990.

[CGG+95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Alg.*, pages 139–149, 1995.

[Com79] D. Comer. The ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, 1979.

[Cor92] Thomas H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[Cor93] Thomas H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan./Feb. 1993.

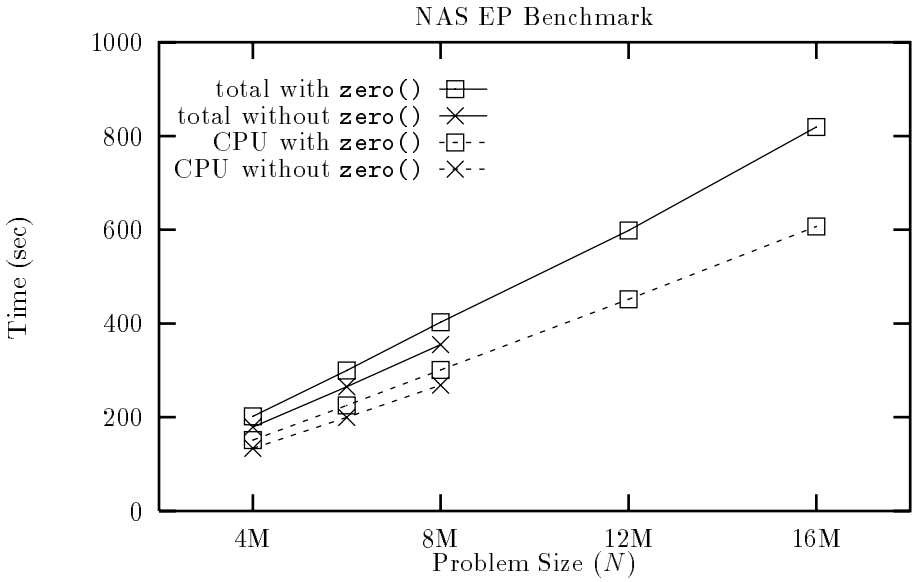

Figure 3: The NAS EP benchmark. Both CPU and overall time including I/O are reported for implementations with and without `zero()`. Larger problem instances are not reported without `zero()`, since there was not enough disk space. For smaller instances, it is clear that the performance overhead associated with using `zero()` is quite small.

[CSW94] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[Fed] Free Software Federation. GNU general public license. Available via FTP from `prep.ai.mit.edu:/pub/gnu/GNUinfo`.

[GP94] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proc. 1st USENIX Symp. on OS Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, Nov. 1994. USENIX Assoc.

[GTVV93] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD Conference*, pages 47–57, 1984.

[Joh95] Michael K. Johnson. *Linux Kernel Hackers' Guide*. Linux Documentation Project, 1995. Available via FTP from `sunsite.unc.edu:/pub/Linux/docs/LDP/` and in printed form from a number of independent Linux distributers.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adapatable, symmetric multikey file structure. *ACM Trans. Databases*, 9(1):38–71, March 1984.

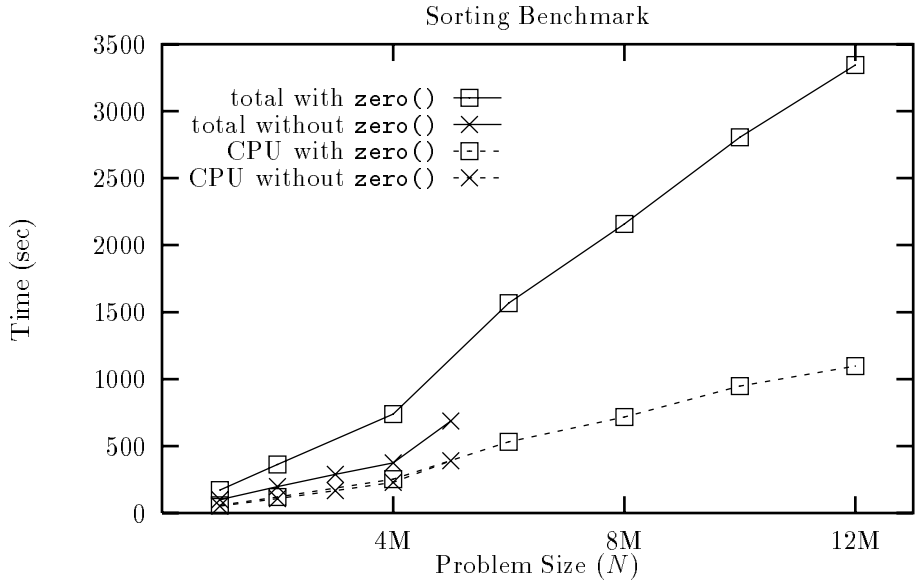

Figure 4: The s24 benchmark. Times are reported as in Figure 3.
The vast majority of the additional time required when `zero()` was used was caused by writing updated metadata to the disk. This cost can likely be reduced by using more advanced metadata management techniques, such as those presented in [GP94].

[NV90] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1990.

[NV93] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, January 1993.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 6(2):1905–1930, July-August 1978.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: a dynamic index for multi-dimensional objects. In *VLDB '87*, 1987.

[Tho78] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 6(2):1931–1946, July-August 1978.

[TRY+87] Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, and Richard Sanzi. A Unix interface for shared memory and memory mapped files under Mach. In *Proceedings of the 1987 Summer USENIX Conference*, pages 53–67, June 1987.

[Ven94] Darren Erik Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, July 1994.

[Ven95] Darren Erik Vengroff. *TPIE User Manual and Reference*. Duke University, 1995. Available via WWW at `http://www.cs.duke.edu:~dev/tpie.html`.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2), 1994.

[VV95] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using tpie. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, San Antonio, TX, October 1995. IEEE.

[Wir95] Lars Wirzenius. *The Linux System Administrators' Guide*. Linux Documentation Project, 1995. Available via FTP from `sunsite.unc.edu:/pub/Linux/docs/LDP/` and in printed form from a number of independent Linux distributers.