Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management

Eric Cooper[1], Scott Nettles[1], and Indira Subramanian[2]

Abstract

We improved the performance of garbage collection in the Standard ML of New Jersey system by using the virtual memory facilities provided by the Mach kernel. We took advantage of Mach's support for *large sparse address spaces* and *user-defined paging servers*. We decreased the elapsed time for realistic applications by as much as a factor of 4.

1 Introduction

Standard ML is a modern functional programming language with a large international community of users [8]. The popularity of the language is due in large part to the Standard ML of New Jersey compiler (SML/NJ), a high-quality, freely available implementation that runs on most UNIX platforms [4].

SML/NJ does not use a stack at runtime; all objects, including activation records, are allocated from the heap. The runtime system uses an efficient variation of copying garbage collection called generational collection [2].

One only has to sit in the same room with a DECstation 3100 running SML/NJ to realize that the performance bottleneck is disk I/O due to paging. The liberal use of heap allocation results in large working sets, and a large amount of paging on workstations with moderate amounts of memory. A particularly common platform in our environment is a DECstation 3100 (based on the MIPS R2000 CPU) with 24 Megabytes of memory, running the Mach 2.5 operating system. With this configuration, compiling a small but significant portion of the SML/NJ compiler takes almost half an hour.

In this paper, we describe two optimizations that we made to the SML/NJ garbage collector in order to improve paging performance. The first is an improved way of managing the old and new areas of the heap, by taking advantage of Mach's support for *sparse address spaces*. The second is an application-specific *external pager* that dramatically reduces disk I/O due to paging, by exploiting the garbage collector's extraordinary knowledge of the state of its address space.

We also present three benchmarks that were used to validate these modifications. The benchmarks show that the combined effect of these two optimizations can improved the performance of SML/NJ on real applications by as much as a factor of 4. For example, the compilation described above now takes only 7 minutes. The results of these benchmarks also suggest some general approaches to the placement of heaps in physical memory.

2 The UNIX version of SML/NJ

One of the goals of the SML/NJ implementors was portability; the runtime system assumes only a 32-bit word size and the UNIX model of an address space (text segment, data segment, and stack segment). The ML heap is placed in the data segment. As the amount of live data increases, the heap size is increased via an operating system call (*sbrk*) that extends the upper boundary of the data

Authors' affiliations: [1]School of Computer Science, [2]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597 and by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0128.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1992 ACM LISP & F.P.-6/92/CA

segment.

The heap is arranged into the following regions:

1. An *old* region, consisting of objects that have survived at least one collection.
2. An *unused* region, into which the old region can expand.
3. A *new* region, consisting of objects created since the last collection.
4. A *free* region, where all new allocation is done.

When the free region is exhausted, a *minor* collection takes place: the live data from the new region is appended to the old region, and execution continues with an empty new region. When the old region reaches half the heap size, a *major* collection takes place, leaving the live data in the upper half of the heap; this is then block-copied to the lower half so that the next minor cycle can begin.

The size and location of each of these regions is controlled by a strategy which is described in detail in Appel's paper [2]. Briefly, the strategy has two parameters:

1. `mem_limit`, the memory use target
2. `ratio`, the target ratio of heap size to live size

After each *major* collection the amount of live data is known, and the total heap size and size of the free region is calculated by

```
if ratio*live_size < mem_limit then
  heap_size = ratio*live_size
else if 3*live_size < mem_limit then
   heap_size = mem_limit
else
   heap_size = 3*live_size

free_size = (heap_size - live_size)/2;
```

After each minor collection `free_size` is recalculated, and the *free* region is placed at the end of the heap.

When we are in the limit that `3*live_size < mem_limit` then `free_size = live_size`. The *free* region fills rapidly, which means in this limit a lower bound on the working set is `2*live_size`.

3 Improving Heap Management

Our first set of improvements involved the management of the heap by the SML/NJ garbage collector. We changed the strategy for determining the sizes of the old and new heaps, and their placement in memory.

The new strategy has three parameters:

1. `mem_lim`, the memory use target
2. `min_new_size`, the minimum new heap size
3. `min_old_incr`, the minimum old heap increment

The basic idea is to keep the portion of SML/NJ's working set used by the heap at or below the memory use target. After a major collection, we set

```
new_heap_size = max
(mem_lim - live_size, min_new_size)
old_heap_size = max
(mem_lim/2, live_size + min_old_incr)
```

The intent is to give the new heap all of the working set that the old heap isn't using, and make the old heap fit in half the working set, so major collections go fast. After a minor collection, we reduce the size of the new heap by however much the old heap grew, while respecting the new heap minimum. With this scheme when the live size is large, the lower bound on the working set size is `live_size + min_new_size`.

We also changed the garbage collector to allow the old heap to drift through memory, by relying on Mach's support for sparse virtual address spaces. In the original SML/NJ scheme, the old heap was copied back to the beginning of the allocation area after every major collection; we now leave it wherever the major collection copied it. When possible, we make this the beginning of the allocation area. This avoids several linear scans of the old heap. The kernel's LRU page replacement policy interacts especially poorly with these scans.

4 Using an External Pager

The SML/NJ runtime system has almost complete knowledge of which pages in the heap are *discardable* and which are *non-discardable*. A *discardable* page is one which contains no useful data, for example, the pages in from-space after a major collection. A system that exploits this knowledge can greatly improve paging performance. For example, when a discardable page is selected for replacement, it need not be written to backing store, because it does not contain any useful data. Similarly, when a nonresident discardable page is referenced, it need not be retrieved from backing store; instead, the existing contents of any physical page can be provided.

The Mach kernel allows the user to create *memory objects* that can be managed by a user-level process, called an *external pager* [10]. The external pager interface enables a user process to manage the paging of memory objects for client applications. The communication between the pager and the kernel occurs via Mach IPC. Through the external pager, a client may create an object and request that the kernel map the object into the client's address space. The pages that constitute the object are subsequently managed

by the external pager. We have implemented an external pager which allows a client to communicate the discardable or nondiscardable state of a page with the pager. For details of the internals of our implementation see Subramanian [9].

4.1 Interaction between SML/NJ and the External Pager

State information about the memory object is maintained in a shared bitmap that is written by the client (SML/NJ) and read by the pager. Each bit indicates whether the corresponding memory object page should be considered discardable or nondiscardable.

The SML runtime sets this bitmap in the following way. During the execution of the user's code, the old, new, and free regions are marked *non-discardable* while all others are marked *discardable*. During minor collection, the pages at the end of the old region which are being copied to are marked *non-discardable*. Before a major collection, the pages of the new and free regions are marked *discardable*. During a major collection the pages being copied to are marked *non-discardable*. After the major collection, the old, new, and free regions are again marked *non-discardable* while all others are marked *discardable*. It should be possible always to keep the pages of the free region marked *discardable*, but the need for rapid allocation makes this difficult as marking pages in the free area *non-discardable* would require trapping into the SML/NJ runtime, an expensive operation given the rate at which memory is allocated.

4.2 Preflushing

Although there is a significant performance gain achieved by using the discardable/non-discardable state of a page, the kernel might choose to replace a non-discardable page even when there are still discardable pages in memory. This is an undesirable consequence of the fact that the external pager is not involved in the page replacement policy, only its execution.

To increase the chances that non-discardable pages will remain resident, the pager can independently ask the kernel to flush one or more discardable pages. But there is a tradeoff here: if we flush pages too eagerly, we may incur unnecessary IPC costs in doing so, and zero-fill costs when we go to reference them again.

We looked at two variations of preflushing. In *pager-initiated* preflushing, the pager uses the arrival of a pageout request from the kernel as an indication that memory is getting tight, and initiates preflushing.

In *user-initiated* preflushing, the client uses a special `FlushDiscardable` operation provided by the pager. Pages are flushed after a major collection.

5 Experimental Method and Results

We ran our benchmarks on a DECstation 3100 workstation with 24 Megabytes of memory, running the Mach 2.5 kernel. This is the maximum memory configuration for this machine. In this configuration there is approximately 11 Mb of free space available for the SML/NJ heap. Performance is particularly poor on this workstation because it has no hardware support for DMA transfers; all disk I/O is done by the CPU. It is nevertheless a very realistic example, since these workstations are currently used throughout our computing facility.

We used three different benchmarks to test the effectiveness of our modifications to SML/NJ. They were chosen to exhibit a wide variety of memory use patterns from paging-bound to cpu-bound. For each of these benchmarks, we tested both the original allocation strategy and the modified one with each possible setting for the pager:

1. no pager (NP)
2. pager only (PO)
3. pager with *pager-initiated* flushing (PF)
4. pager with *user-initiated* flushing (PU)

We also studied the effect of whether the heap was copied back to the beginning of memory, or not. By instrumenting the pager we were able to get details of the paging activity.

The optimal settings of the heap parameters described above depend on the particular benchmark and machine configuration. We have not yet incorporated an algorithm to adjust these dynamically; instead, we used a brute-force search of the parameter space to find the minima. We performed this search for each major variation described above. When testing copy-back we used the parameters found for the major variations, so these results represent upper bounds. In general we found that the original heap sizing strategy was less sensitive to settings of its parameters than the modified one.

In each of the three following sections we discuss a specific benchmark. We first discuss the benchmark in general, including a plot of the live size vs. total memory allocated. Total memory allocated represents a simple kind of pseudotime. These plots allow one to understand the demands of the benchmark for memory. Next, we present the basic results for each of the allocation schemes with all possible pager parameters. These are wall clock times in seconds. Finally we present the results of varying copy-back. For the original scheme this means turning copy-back off, while for the modified scheme it means turning copy-back on. Details such as the settings of the memory parameters, values of clock and system time, paging statistics, etc. are found in the appendix.

5.1 The Compiler Benchmark

The compiler benchmark compiles a portion of the SML/NJ compiler. A plot of the live data size vs. the total memory allocated is shown in figure 1. For much of its execution, this benchmark has more than 5 Mb of live data. For significant portions of its execution, this benchmark is paging-limited. The saw tooth structure indicates periods when it is building large intermediate data structures, and then discarding them. We speculate that these occur during the compilation of individual functions or modules, with storage being freed after code generation.

Figure 2 shows the times for the various basic parameters. The new allocation strategy is always faster than the original, some times by more than a factor of 2.5. The paging statistics indicate that less paging is occurring. For both allocation strategies, use of the pager results in significant performance gains, with many of the paging events no longer accessing the disk. Adding flushing is also quite beneficial. For the new allocation strategy, both forms of flushing result in approximately 30% improvements, and similar paging statistics. For the original allocation scheme pager-initiated flushing seems to be preferred. It resulted in less paging activity of all kinds, even though, interestingly, it flushed approximately the same number of pages. The overall improvement is almost a factor of four.

Figure 3 shows the time for the original strategy without copy-back and Figure 4 shows the time for the modified strategy with it. We found the results for the no pager case to be quite surprising. Using copy-back seems to improve the results by a factor of two for the original strategy and a significant amount for the modified one as well. The detailed paging statistics show that for the original scheme, the no copy-back case does almost twice as much paging as the copy-back case. Since executions are basically paging limited, this explains the factor of two. We believe this is because for most of its execution the total live size is close to half the available memory. When copy-back is performed, this forces the available pages into a contiguous region which then become the old and new area. This essentially pre-pages the free region. When copy-back is not performed, the free region is located in a portion of memory which has not been touched recently. Before memory can be allocated to the free region, it must be taken from the old from-space and paged out.

When the pager is used, the cost of moving pages from the old from-space to the free region drops, since these pages no longer have to be written to disk. Adding flushing further changes the situation since they are flushed. This results in the expected behavior for copy-back.

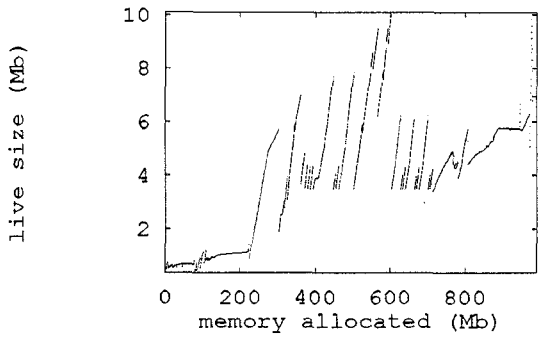

Figure 1: live size vs. total allocation for compiler

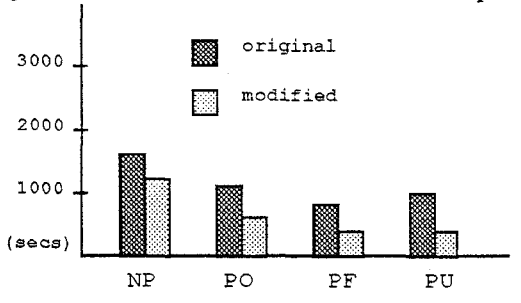

Figure 2: results for compiler

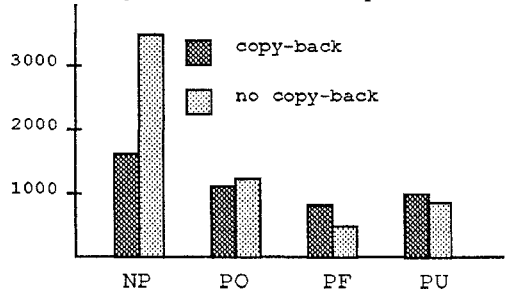

Figure 3: original allocation strategy for compiler

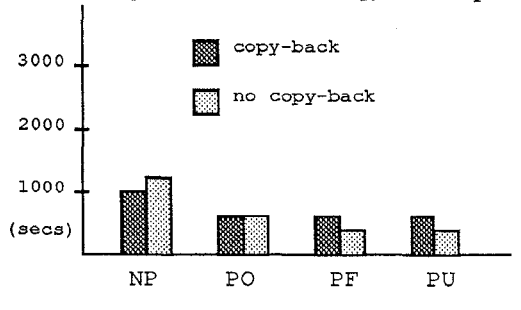

Figure 4: modified allocation strategy for compiler

5.2 The Sort Benchmark

The sort benchmark performs a merge sort based on divide and conquer. The divide steps are implemented by spawning futures, which are in turn implemented using a simple threads package [5]. A plot of the live data size vs. the total memory allocated is shown in figure 5. The computation starts out cpu-bound, but becomes very paging-bound. At its largest, the live data alone exceeds the available free memory. The fact that we do not see a divide phase during which time the live data decreases is probably an artifact of using allocation as a time metric. The divide phase does almost no allocation, and therefore is hidden at the end of the plot.

Figure 6 shows the run times for the two allocation strategies and each of the pager parameters. The new allocation scheme performs approximately a factor of two better than the original. The paging statistics indicate this corresponds quite closely to the difference in paging activity. The most surprising feature of these results is the fact that for the new allocation scheme, the pager actually performs worse than without the pager. We are at a loss to explain this; perhaps it is an artifact of the minimization process. Pager-initiated flushing is always better than user-initiated even though it flushes approximately the same number of pages. Altogether there is a performance improvement of approximately four.

Figure 7 shows the time for the original strategy with copy-back turned off. There is little difference from the original for the no pager case. We believe that this is due to a balance between the prepaging of the new area and the extra paging due to copying during copy-back. In general this case is probably dominated by the cost of paging during the phase of execution when the live size is very large. When the pager is used, the copy-back starts to become less attractive. In this case we have made it less expensive to transfer pages from the old from-space to the new area, so pre-paging is less helpful, but copy-back still incurs the cost of paging during copies. This effect becomes even more pronounced as we add flushing, with the kernel flushing case approaching the best performance of the new allocation strategy.

Figure 8 shows the time for the modified strategy with copy-back turned on. In this case copy-back is clearly undesirable, with the performance penalties ranging from factors of two to as much as a factor of three. Examining the worst case, we see that there are three times as many reads and page outs which go to disk, and fifteen times as many write faults which go to disk. The other cases show similar although less dramatic results.

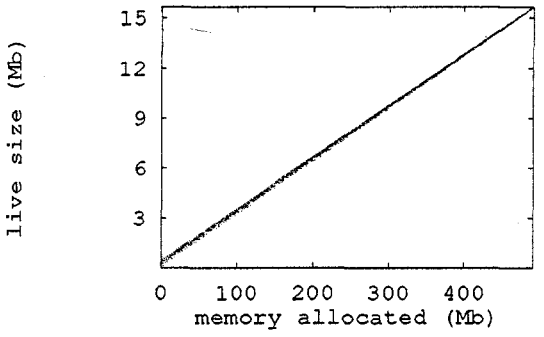

Figure 5: live size vs. total allocation for sort

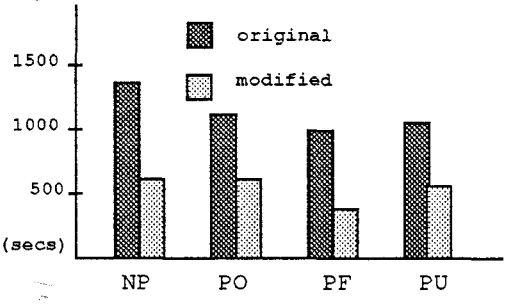

Figure 6: results for sort

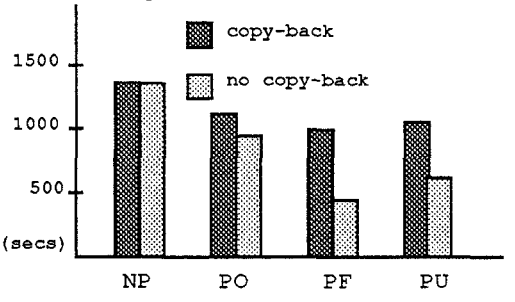

Figure 7: original allocation strategy for sort

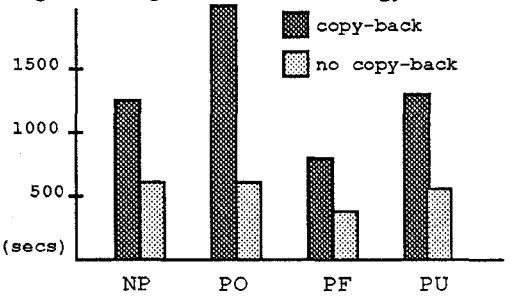

Figure 8: modified allocation strategy for sort

5.3 The Primes Benchmark

The primes benchmark finds primes using the sieve of Eratosthenes, implemented in a simple interpreted lazy language. The interpreter is written in SML. As figure 9 shows, this benchmark is completely CPU limited. Thus it serves as a good test of whether our techniques carry any substantial penalty for applications which do not page.

Figure 10 shows the run times for the two allocation strategies and each of the pager parameters. They show almost no variation between either allocation strategy or pager parameter setting. Examination of the paging statistics also show almost identical results. In the case of *user-initiated* flushing, the fact that flushes occur does cause some of the paging statistics to vary, but without any noticeable effect on the times. This gives weight to our belief that paging events which do not access the disk have little impact on run times.

Figure 11 shows the time for the original strategy with copy-back turned off. Here we do see some penalties for not using copy-back, probably for the same reasons given above. We performed a minimum search on the cases which showed variation. This eliminated the overhead due to not doing copy-back. These minima had settings for `mem_limit` which were substantially smaller than for the copy-back case. With these smaller settings, no paging occurs.

Figure 12 shows the time for the modified strategy with copy-back turned on. In this case we see only small variations which we do not believe to be significant.

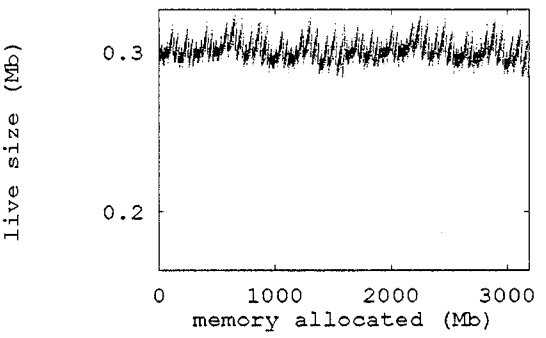

Figure 9: live size vs. total allocation for primes

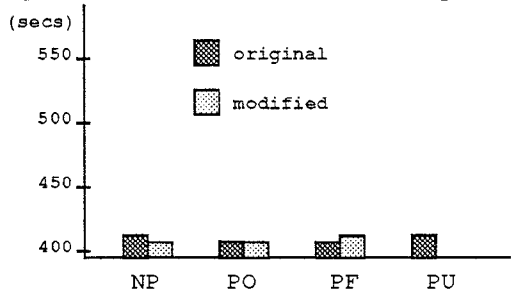

Figure 10: results for primes

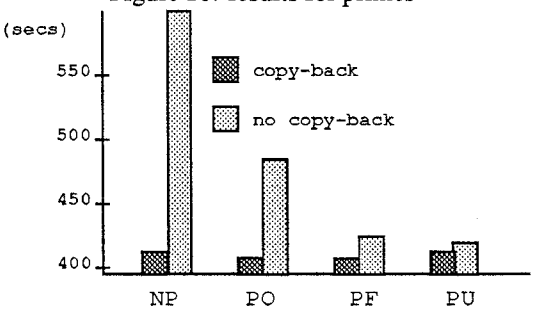

Figure 11: original allocation strategy for primes

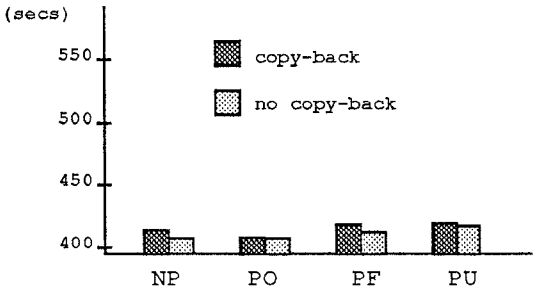

Figure 12: modified allocation strategy for primes

6 Summary

Our modifications show impressive results. Both the new allocation strategy and the external pager are effective at reducing the amount of paging, and thus at speeding SML/NJ programs. For programs which do not page they do no harm. We draw several lessons from these results.

One set of lessons applies to any OS environment and has to do with careful management of physical memory. Our improved allocation strategy shows that significant gains can be had by simple consideration of working set size issues. The results of varying copy-back shows further how use of physical memory resources can affect performance. We believe that if we had placed the free region so as to take advantage of the prepaging of new space, we would have reaped the benefits of copy-back without the deficits.

The other set of lessons are OS specific. First, the idea of discardable pages is a useful one, especially for garbage collected languages. In addition we suspect that a simpler interface based on flushing might show much of the improvements of discardable pages, without the need for tightly coupled communication between the pager and the application. Finally we strongly support the idea of letting the user gain control of paging. While it may turn out to be more efficient to support discardable pages in the kernel, the flexibility and implementation ease of being able to do it outside the kernel was key. If we had only the option of an in-kernel implementation, we would probably not have tested the idea at all.

7 Related Work

The relationship between garbage collection and virtual memory support has been explored by a number of researchers. One of the early motivations for copying garbage collection [6] was its improved locality of reference compared to the mark and sweep technique. More recently, virtual memory hardware has been used to allow concurrent garbage collection [3].

There are two main areas of related work that complement our approach.

We have not experimented with any form of adaptive control algorithm to vary our heap size parameters dynamically. Alonso and Appel [1] have done such experiments: they observed that an application like SML/NJ could expand or contract its working sets to match the amount of real memory available to it, if only it had access to such information. They developed a centralized *advice* server to provide client processes with this information. Our external pager could equally well double as an advice server.

The Mach interface to external pagers does not allow user control over the page replacement policy, only what is done with a page once it has been chosen for replacement. This is unfortunate: in the case of SML/NJ, we would like to indicate that any discardable page should be chosen for replacement before any non-discardable page. McNamee and Armstrong [7] have extended the Mach external pager interface with what they call *page replacing memory objects*. In their system, the kernel asks the pager to relinquish some number of pages; the pager can then use any application-specific policy to determine which ones. The cost of this approach is more kernel-to-pager IPC traffic; whether it would be outweighed by the reduction in paging must await further research.

8 Future Work

The current pager uses a general interface to the user program. Analysis of our current results should allow us to tailor the pager specifically to SML/NJ, allowing it to make more intelligent decisions about how to manage the address space. This is clearly one of the next significant step to take. We would also like to explore the techniques discussed in related works.

Managing discardable pages with an external pager has been shown to yield good results, especially when using preflushing techniques. However, it is evident that having the information about discardable pages within the kernel would have certain advantages: Performing too many preflushes is likely to impede system performance; in the case of in-kernel implementation, knowledge of physical memory availability would enable the kernel to flush a discardable page only when necessary, thereby avoiding unnecessary zero-fills

It is possible to reduce the number of zero-fills without compromising security. For example, in Mach, we could let the kernel keep track of which memory object previously owned a page in the free list. When a task page-faults on a discarded page, the kernel could return a page in the free list which had previously belonged to the memory object, thus avoiding the need to zero-fill the page. It is our belief that systems such as SML/NJ which page heavily against their own pages and generate many discardable pages will benefit from this approach. We are in the process of evaluating an in-kernel implementation of discardable page management under Mach version 3.0.

9 Conclusion

The idea of using the garbage collector's knowledge to distinguish between discardable and non-discardable pages is a simple one. Our main contribution is to show how Mach's flexible virtual memory system can exploit this idea with great practical benefit.

Acknowledgments: We would like to thank Peter Lee for a substantial fraction of the several CPU months we needed for our benchmarking. Anurag Acharya and Greg Morrisett provided valuable feedback at various times through out the work, as did the whole CMU SML community. We would like to thank Penny Anderson, Ellen Siegel and the Venari group for proof reading.

References

[1] Rafael Alonso and Andrew W. Appel.
An advisor for flexible working sets.
In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 153–162, May 1990.
Also published as *Performance Evaluation Review*, 18(1).

[2] Andrew W. Appel.
Simple generational garbage collection and fast allocation.
Software—Practice & Experience, 19(2):171–183, February 1989.

[3] Andrew W. Appel, John R. Ellis, and Kai Li.
Real-time concurrent collection on stock multiprocessors.
In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.
Also published as *SIGPLAN Notices*, 23(7).

[4] Andrew W. Appel and David B. MacQueen.
A Standard ML compiler.
In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.
Volume 274 of *Lecture Notes in Computer Science*.

[5] Eric C. Cooper and J. Gregory Morrisett.
Adding threads to Standard ML.
Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[6] Robert R. Fenichel and Jerome C. Yochelson.
A LISP garbage collector for virtual-memory computer systems.
Communications of the ACM, 12(11):611–612, November 1969.

[7] Dylan McNamee and Katherine Armstrong.
Extending the Mach external pager interface to accomodate user-level page replacement policies.
In *Proceedings of the USENIX Mach Workshop*, pages 17–29. USENIX Association, October 1990.

[8] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[9] Indira Subramanian.
Managing discardable pages with an external pager.
In *USENIX Mach Symposium*, pages 77 – 85. USENIX Association, November 1991.

[10] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron.
The duality of memory and communication in the implementation of a multiprocessor operating system.
In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.
Published as *Operating Systems Review*, 21(5).

10 Appendix

	NP	PO	PP	PU
mem_lim (Kb)	14000	9000	9000	16000
ratio	1000	7	7	4
user time (sec)	300.8	344.4	352.1	335.3
system time (sec)	59.9	50.7	69.1	82.8
clock time	27:21	19:36	13:32	17:01
read faults zero-filled	19	7	903	1037
read faults not read	0	98	4938	7638
read faults read	6797	8075	4798	8544
write faults zero-filled	6069	6619	4663	4866
write faults not read	0	11806	17550	24662
write faults read	18648	7936	3487	3819
page-out discarded	0	14660	428	9449
page-out written	26883	14890	8349	13016
pages flushed	0	0	24109	24338

Figure 13: Compiler original strategy copy-back on

	NP	PO	PP	PU
mem_lim (Kb)	3000	7000	6000	11000
ratio	4	4	100	10
user time (sec)	194.2	186.1	196.4	166.8
system time (sec)	33.8	44.2	46.1	52.6
clock time	23:13	18:35	16:49	17:48
read faults zero-filled	14	13	17	17
read faults not read	0	27	145	124
read faults read	10818	10736	11798	17424
write faults zero-filled	8429	8245	9499	10412
write faults not read	0	5224	10955	17393
write faults read	11585	5901	6511	9037
page-out discarded	0	5553	99	5767
page-out written	25682	19492	21681	30393
pages flushed	0	0	13276	14296

Figure 14: Sort original strategy copy-back on

	NP	PO	PP	PU
mem_lim (Kb)	12000	12000	14000	14000
ratio	1000	1000	100	1000
user time (sec)	404.4	398.2	397.6	396.6
system time (sec)	4.6	6.7	6.0	9.8
clock time	6:53	6:49	6:47	6:52
read faults zero-filled	5	6	868	897
read faults not read	0	0	0	1132
read faults read	0	0	0	0
write faults zero-filled	2995	2994	2632	2603
write faults not read	0	0	0	1478
write faults read	0	0	0	0
page-out discarded	0	0	0	0
page-out written	0	0	0	0
pages flushed	0	0	0	3409

Figure 15: Primes original strategy copy-back on

	NP	PO	PP	PU
mem_lim (Kb)	14000	9000	9000	16000
ratio	1000	7	7	4
user time (sec)	284.0	297.1	310.0	293.2
system time (sec)	85.1	94.3	84.1	81.0
clock time	59:35	20:35	8:49	13:20
read faults zero-filled	32	27	19	1950
read faults not read	0	265	360	9605
read faults read	11942	9397	1163	15009
write faults zero-filled	11020	10155	10152	8486
write faults not read	0	36532	41923	34454
write faults read	39083	4303	949	4483
page-out discarded	0	41668	958	11172
page-out written	57329	14633	2124	20046
pages flushed	0	0	47715	38144

Figure 16: Compiler original strategy copy-back off

	NP	PO	PP	PU
mem_lim (Kb)	3000	7000	8000	11000
ratio	4	4	100	10
user time (sec)	158.2	159.9	138.5	146.8
system time (sec)	45.4	63.5	38.5	54.4
clock time	23:43	16:03	7:15	10:37
read faults zero-filled	59	42	44	1365
read faults not read	0	318	236	3906
read faults read	7232	9494	3279	4560
write faults zero-filled	11947	13032	11625	9195
write faults not read	0	12920	11284	11366
write faults read	10620	1208	413	2249
page-out discarded	0	17443	165	2700
page-out written	25843	15579	6955	10168
pages flushed	0	0	15879	15968

Figure 17: Sort original strategy copy-back off

	NP	PO	PP	PU
mem_lim (Kb)	12000	12000	12000	12000
ratio	1000	1000	1000	1000
user time (sec)	404.4	403.0	398.2	405.0
system time (sec)	4.6	30.2	6.7	26.7
clock time	6:53	10:02	6:49	8:11
read faults zero-filled	5	9	6	7
read faults not read	0	0	0	5
read faults read	0	45	0	42
write faults zero-filled	2995	4459	2994	4461
write faults not read	0	0	0	2922
write faults read	0	2940	0	16
page-out discarded	0	0	0	2951
page-out written	0	3501	0	574
pages flushed	0	0	0	0

Figure 18: Primes original strategy copy-back off

	NP	PO	PP	PU
mem_lim (Kb)	3500	3000	10000	0
min_new_size (Kb)	2500	500	1000	500
min_old_incr (Kb)	3000	3500	5500	5000
user time (sec)	297.8	302.4	259.0	280.5
system time (sec)	39.5	52.8	55.4	54.3
clock time	21:02	10:03	7:13	7:11
read faults zero-filled	29	47	36	40
read faults not read	0	274	330	444
read faults read	5349	3047	540	585
write faults zero-filled	6925	7910	8141	8201
write faults not read	0	19292	28783	29340
write faults read	16562	2252	765	887
page-out discarded	0	23196	537	155
page-out written	25110	5644	1435	1682
pages flushed	0	0	32884	34798

Figure 19: Compiler modified strategy copy-back off

	NP	PO	PP	PU
mem_lim (Kb)	0	9000	6000	9000
min_new_size (Kb)	6000	5000	7000	5000
min_old_incr (Kb)	10000	5000	7000	5000
user time (sec)	126.5	129.2	110.4	132.4
system time (sec)	34.1	46.0	31.0	42.2
clock time	9:55	10:47	6:09	8:59
read faults zero-filled	242	150	2428	132
read faults not read	0	111	1797	94
read faults read	4944	6678	4162	5136
write faults zero-filled	10783	10697	5630	10715
write faults not read	0	9358	5340	9285
write faults read	3113	610	2641	520
page-out discarded	0	12419	428	2917
page-out written	15135	11246	9509	8529
pages flushed	0	0	8416	10522

Figure 20: Sort modified strategy copy-back off

	NP	PO	PP	PU
mem_lim (Kb)	0	2500	0	8500
min_new_size (Kb)	4000	4500	5000	4500
min_old_incr (Kb)	4000	500	1000	4000
user time (sec)	397.2	399.1	398.7	394.1
system time (sec)	5.8	6.2	6.7	14.2
clock time	6:44	6:48	6:49	6:55
read faults zero-filled	26	13	515	1529
read faults not read	0	0	0	738
read faults read	0	0	0	0
write faults zero-filled	3182	2703	2417	4708
write faults not read	0	0	0	3121
write faults read	0	0	0	0
page-out discarded	0	0	0	0
page-out written	0	0	0	0
pages flushed	0	0	0	7178

Figure 21: Primes modified strategy copy-back off

	NP	PO	PP	PU
mem_lim (Kb)	3500	3000	10000	0
min_new_size (Kb)	2500	500	1000	500
min_old_incr (Kb)	3000	3500	5500	5000
user time (sec)	347.7	336.5	287.7	312.8
system time (sec)	35.4	26.9	71.6	78.1
clock time	16:51	11:26	11:11	10:24
read faults zero-filled	15	20	35	15
read faults not read	0	17	221	189
read faults read	3736	2294	1038	591
write faults zero-filled	4759	4790	6212	4788
write faults not read	0	2002	37656	31406
write faults read	7670	3183	3189	2488
page-out discarded	0	2941	459	0
page-out written	12372	5505	4537	3079
pages flushed	0	0	39712	33506

Figure 22: Compiler modified strategy copy-back on

	NP	PO	PP	PU
mem_lim (Kb)	0	9000	6000	9000
min_new_size (Kb)	5500	5500	7500	5000
min_old_incr (Kb)	10500	5500	6500	5000
user time (sec)	111.7	146.0	128.7	161.8
system time (sec)	31.6	62.4	41.9	54.9
clock time	20:31	32:50	13:15	21:46
read faults zero-filled	39	36	1442	42
read faults not read	0	80	2314	129
read faults read	8364	18564	7653	12482
write faults zero-filled	7695	7480	6607	6752
write faults not read	0	12006	6201	11950
write faults read	9632	11583	3502	7548
page-out discarded	0	12086	245	3848
page-out written	21181	31678	13317	21936
pages flushed	0	0	10386	9208

Figure 23: Sort modified strategy copy-back on

	NP	PO	PP	PU
mem_lim (Kb)	0	2500	0	8500
min_new_size (Kb)	4000	4500	5000	4500
min_old_incr (Kb)	4000	500	1000	4000
user time (sec)	405.5	401.3	406.6	398.7
system time (sec)	5.3	4.5	6.6	10.8
clock time	6:55	6:48	6:59	6:58
read faults zero-filled	31	20	440	592
read faults not read	0	0	0	1199
read faults read	0	0	0	0
write faults zero-filled	2175	1529	1458	2718
write faults not read	0	0	0	1738
write faults read	0	0	0	0
page-out discarded	0	0	0	0
page-out written	0	0	0	0
pages flushed	0	0	0	3328

Figure 24: Primes modified strategy copy-back on