
Virtual Memory Support for Multiple Pages


Yousef A. Khalidi
Madhusudhan Talluri
Michael N. Nelson
Dock Williams

SMLI TR-93-17

September 1993

Abstract:

The advent of computers with 64-bit virtual address spaces and giga-bytes of physical memory will provide applications with many more orders of magnitude of memory than is possible today. However, to tap the potential of this new hardware, we need to re-examine how virtual memory is traditionally managed. We concentrate in this note on two aspects of virtual memory: software support for multiple page sizes, and memory management policies tuned to large amounts of physical memory. We argue for the need to examine these areas, and we identify several questions that need to be answered. In particular, we show that providing support for multiple page sizes is not as straightforward as may initially appear.

 **Sun Microsystems**
Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
yousef.khalidi@eng.sun.com
madhusudhan.talluri@eng.sun.com
michael.nelson@eng.sun.com
dock.williams@eng.sun.com

Virtual Memory Support for Multiple Page Sizes

Yousef A. Khalidi, Madhusudhan Talluri, Michael N. Nelson, Dock Williams

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043 USA

1 Introduction

Virtual memory implementations in current systems such as UNIX[®] [5], VMS[®] [3], NT[™] [13], MACH[®] [2], and CHORUS[®] [4], share two basic assumptions regarding physical memory management:

- There is one page size. This one size may be a multiple of the MMU page size, but there is only one size and it is typically in the range of 512-8K bytes.
- Physical memory is not very large, typically somewhere in the range of 4M-256M bytes. Physical memory replacement algorithms are normally tuned for the common sizes of 16M-64M bytes.

It is not hard to see that these assumptions will no longer be valid for a class of machines in the near future. We believe that to tap the potential of computers with large amounts of memory requires new software implementation techniques, rather than just porting the existing operating systems to future machines.

In this note, we examine two hardware features of future machines that violate basic assumptions of current virtual memory implementations:

- Hardware support for multiple pages sizes.
- Very large physical memory.

2 Multiple page sizes

2.1 Motivation

A Translation Look-aside Buffer (TLB) is a cache of virtual-to-physical address translations, and is typically used to reduce the average address translation time. When a

required translation is not in the TLB, a software (e.g., R4000[®] [6]) or a hardware miss handler (e.g., Super-SPARC[™] [7]) is executed to enter the translation in the TLB.

Technological and architectural trends are leading towards larger main memory sizes and programs with larger working sets, but with the TLB size remaining relatively small. For reasons stated in [1] and elsewhere, the TLB size is not expected to increase at the same rate as main memory size, yet the amount of memory mapped by the TLB is an important factor in determining performance. Therefore, there is a need to make the TLB map larger working sets. Otherwise, TLB miss handling may become a performance bottleneck.

One way to increase the amount of memory mapped by the TLB is to increase the page size. This approach may be feasible for modest increases in page sizes (e.g., from 4K to 8K or even 16K bytes). In general, however, this approach causes more internal fragmentations, larger protection boundaries, and forces the system to do all operations, such as copy-on-write, zero-fill on demand, and paging I/O at the new large page size.

Another way to increase TLB coverage is by using multiple-page sizes, where each TLB entry may map a page of variable size. For example, large contiguous regions in a process' address space, such as program text, may be mapped using a small number of large pages (e.g., 64K byte pages) rather than a large number of small pages (e.g., 4K byte pages), while thread stacks may continue to be mapped using the small page size.

The case for hardware support for multiple page sizes has been argued by others [1], and there are now several microprocessor architectures that support multiple page

sizes, including R4000 [6], Alpha [8], and SuperSPARC [7]. For example, the R4000 supports seven page sizes (from 4K to 16M bytes) using a 48-entry fully associative TLB.

In general, there are two classes of applications that can benefit from multiple page sizes:

- The operating system kernel and devices such as frame buffers. In addition to the microprocessors listed above, many other architectures have some limited support for handling these specialized cases, including the PA-Risc™ 1.1 [9] and i860™ [10].
- General applications, including multiprogrammed job mixes, and applications with large working sets such as numerical analysis code.

Mapping the kernel, or specialized devices such as frame buffers using large mappings, is relatively straightforward. Moreover, adding such functionality to existing systems is a simple engineering exercise that affects limited portions of the operating system. In this note, we are interested in providing multiple page size support for general application code.

2.2 Utilizing New Hardware

New microprocessors that support multiple page sizes will be under-utilized if the software does not provide support for multiple page sizes. A simulation study was conducted to compare three different TLB designs, given the same chip area. The bar graph in Figure 1 shows the TLB miss ratios for four large programs and three alternative TLB configurations. The first bar shows the miss ratio (% of memory references that missed in the TLB) for a 64-entry fully-associative TLB with the software using only 4K pages. The second bar shows the miss ratio for a 256-entry 2-way set-associative TLB again using only 4K pages. The third bar shows the same configuration as the first bar but the software is using both 4K and 32K pages (the policy for assigning pages is described in [1]).

Current microprocessor hardware is being designed assuming that the software will use the large page sizes (third bar). However, current operating systems do not support multiple page sizes and end up using only one page size although the TLB can support multiple sizes (first bar). If the hardware engineers knew that the large

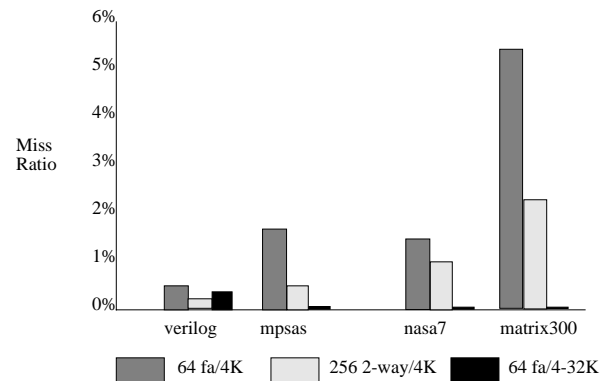


Figure 1. Miss ratios for different TLB

mappings would not be used, a larger set-associative TLB could have been an alternative design (second bar).¹

The figures show that unless large mappings are used, the hardware is being under-utilized. The performance using only small pages is worse than the simpler 2-set associative hardware designs. By using multiple page sizes, there is a *potential* for very low TLB miss ratio. The questions to ask are:

- How would alternative TLBs supporting only a single page size perform, compared to a TLB supporting multiple page sizes?
- What is the overhead in the operating system to support multiple page sizes?

2.3 Why is it Hard?

Current VM systems are not suited for supporting multiple page sizes as the knowledge of one page size is ingrained in the virtual memory code. As we will show, adding support for multiple page sizes raises many new issues regarding how *physical* memory is managed, as well as affecting common VM and file system optimizations such as read-ahead, clustering [16], and copy-on-write.

It is important to note that supporting multiple page sizes affects both the machine dependent *and* independent portions of the system. That is, the clean separation of machine independent and dependent code in systems such

1. This assumes that a fully-associative TLB takes four times the chip area as a set-associative TLB of the same number of entries, with access time being approximately equal.

as MACH, SunOSTM [5], and CHORUS is not enough to mitigate the difficulties of supporting multiple page sizes.

2.4 What Needs to be Done?

2.4.1 Choosing a Page Size

Current VM interfaces do not have the notion of multiple page sizes. Given a choice of more than one page size, who should decide the page size for a given mapping? The user application, the compiler, and the operating system are potential candidates. Should the user even know that a particular mapping can be done using a large page size instead of a number of small pages? Can the compiler make use of multiple-page-size knowledge in arranging the program's data structures?

In general, the system can use large pages to map any mapped region that is big enough. However, as we will show below, in many cases the system may not be able to satisfy a request for a large page mapping, and instead may resort to using a number of small mappings. Therefore, at best, the notion of multiple page sizes should be exported to the user/compiler as a hint only.

2.4.2 Arranging VM Data Structures

Introducing support for multiple page sizes complicates many VM data structures. Consider, for instance, the basic page frame descriptor. Traditionally, VM systems represent each physical page with a descriptor. The descriptor is normally linked on a number of lists, contains (or implicitly indicates) the page frame number, and includes other state information (e.g., locks, modified and referenced bits).

With multiple page sizes, a particular page frame descriptor may represent one of several page sizes. Moreover, at any point in time, a page frame may need to be viewed simultaneously as a distinct small page and as part of a larger page (e.g., when two mappings are established to the same memory object: one small and one large). Traditionally, a hash list of all page frames in the system is built and is accessed by some page frame identifier (e.g., in SunOS, by a vnode-offset pair). With multiple page sizes, arranging and accessing the hash list can become very complex.

2.4.3 Managing Physical Memory

Perhaps the biggest issue with adding support for multiple page sizes is how to manage physical memory. Micropro-

cessors that support multiple page sizes normally impose the following alignment restriction: a mapping of size B bytes is defined to start at a virtual address that is a multiple of B and a physical address that is a multiple of B , where B is a power of 2. Note that the physical address specifies a contiguous physical page of size B .

These constraints are due to what the hardware can support efficiently in TLBs. Defining a page, as above, allows the hardware to be based on bit-selection. Supporting unaligned virtual/physical addresses requires hardware adders/comparators that are more complicated. All existing microprocessor TLBs that we are familiar with require these alignment restrictions.

Therefore, to support multiple page sizes, there is now a need for managing *physical* memory. For example, the VM system will need to structure its "free" and "clean" page lists into several lists of the supported page sizes (e.g., 8K, 64K, 512K bytes). A physical memory allocator, (e.g., one based on the buddy system) is needed to maintain these lists. When a particular mapping to a large page is requested (e.g., as a hint from the user or from the higher level of the VM), a clean page of the required size must be located. If a page of the appropriate size is not found, the system may opt for using smaller mappings, or may attempt somehow to coalesce smaller pages into a big page.

There are many important questions that need to be answered with respect to physical memory management:

- Can physical memory become so fragmented that no large mapping requests can be accommodated? When and how should the allocator coalesce memory? Will it be worthwhile to copy memory around to create large pages?
- How is page-replacement affected? Should the system favor large pages over small pages? When should a large page be broken into a number of small pages?
- Will managing the free lists according to size cause the system to page more? If it causes extra paging, it's probably not worth the effort! It is important to note that for a given application, one extra I/O operation may wipe out all the advantages of better TLB coverage.
- How does arranging the free lists according to size affect page-coloring [14] algorithms?

2.4.4 Using Large Mappings

Another question to answer is: when should the decision to use a large mapping be made? Should the system allocate and populate a large physical page on the first fault, or should it delay the decision until a later point in time?

A similar question affects copy-on-write and zero-fill-on-demand mappings. The system should probably wait until “enough” small pages have been populated before *promoting* [1] the mapping into one using a large page.

2.4.5 Interaction Between Different Mappings

When using one page size only, all mappings to the same memory object are done using the same page size. With use of multiple page sizes, it is possible that one mapping may use a small page size to map a small part of a memory object, while a second mapping may use a large page size to map a large part of the same memory object. If the system is not careful, the first mapping (which uses small pages) may preclude using large pages for the second mapping.

For example, if one process maps the first 8KB of a 1MB sized file, the VM system may choose a free 8KB physical frame for the mapping. There are no particular alignment requirements on this 8KB physical page. If a second process then requests say a 1M mapping to the same file, then the second mapping cannot be done using a large page size because the VM system already allocated an unaligned 8K page for the first mapping. Even if the 8K page was aligned correctly on a 1MB boundary, the rest of the physical memory after the 8KB page may not be free.

2.4.6 Re-examining VM/FS Optimizations

What are the effects of supporting multiple page sizes on VM and file system optimizations such as read-ahead, and clustering? What about copy-on-write and zero-fill-on-demand handling that was mentioned before?

3 Page replacement

Current commercial high-end machines can support gigabytes of physical memory (e.g., SparcCenter 2000 and CHALLENGE™), and newer machines are expected to reach tera-bytes of physical memory. Current workstations support up to 0.5 gigabytes of memory or more (e.g., SPARCStation-10). The question is how will large amounts of physical memory, especially when coupled

with support for multiple pages sizes, affect page replacement?

Virtual memory systems traditionally keep track of a reference bit per page. This bit is either provided by the hardware or simulated in software. The reference bit is usually used by the VM system to implement page replacement algorithms such as “clock” [17] and Sampled Working Set [18] algorithms. Periodically, the VM system examines all pages in the system, resetting the reference bits, and updating page usage statistics.

It is not clear how useful reference bits will be for very large machines. In particular, clock-based algorithms such as the ones used in many UNIX variants are not appropriate for large amounts of memory (how much time does it take to sweep a 1GB of physical memory? 10GB?).

There is a large body of work from the sixties and seventies in the area of page replacement and paging (see [18] and [19] for references). Perhaps now it is time to re-examine the basic assumptions of prior studies in light of very large physical memory and multiple page sizes. Perhaps we need to reconsider two of the simplest page replacement algorithms that were dismissed in the past: random replacement, and first-in first-out replacement.

One final issue is how the page replacement algorithm interacts with support for multiple page sizes. Should we attempt to bias the algorithm toward large or small pages? When should we break a large page into a set of small pages? Or should we replace large pages completely? By using a simple replacement algorithm such as random replacement, perhaps we can simplify the interaction of page replacement with multiple page sizes.

Others have recognized the need to re-examine page replacement algorithms in large memory machines. Wood and Katz argued for getting rid of the modified bit [12], though we do not necessarily agree with their conclusions. Harty and Cheriton [15] describe an interface exported by the VM system to external pagers that can be used to control physical memory. Harty and Cheriton recognize that future computers will have giga-bytes of memory and advocate providing more control over memory to external pagers. (They also state that their interfaces are useful for providing support for multiple page sizes, though we believe that the interfaces they propose are not sufficient for that purpose.)

4 Conclusions

We believe that new virtual memory implementation techniques are required to tap the potential of newer microprocessors. We posed several questions in this note regarding multiple page size support and new page replacement algorithms.

Few would argue that we need to re-examine current techniques for page replacement in light of machines with very large physical memory. The issue of support for multiple page sizes, however, is more complex. The gains of better TLB coverage can be very small, and the overhead of providing multiple page size support is unknown. As we mentioned before, in some situations, even one extra I/O operation may wipe out the advantages of extra TLB coverage.

Since microprocessors are now designed with hardware support for multiple page sizes, it behooves those of us in the operating system community to investigate how useful hardware support for multiple page size is, and how the operating system can make use of this hardware feature. The operating system should either provide support for multiple page sizes, or the hardware should be redesigned to support a single page size more efficiently.

References

- [1] Talluri, Madhusudhan, Shing Kong, Mark D. Hill, and David A. Patterson. "Tradeoffs in Supporting Two Page Sizes." *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992): 415–424.
- [2] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *IEEE Transactions on Computers* 37, no. 8 (August 1988): 896–908.
- [3] Kenah, Lawrence J., Ruth E. Goldenberg, and Simon F. Bate. *VAX/VMS Internals and Data Structures*. Digital Press, 1988.
- [4] Abrosimov, Vadim, Marc Rozier, and Marc Shapiro. "Generic Memory Management for Operating System Kernels." *12th Symposium on Operating System Principles (SOSP '89)* (1989): 123–136.
- [5] Gingell, Robert A., Joseph P. Moran, and William A. Shannon. "Virtual Memory Architecture in SunOS." *Proceedings of Summer '87 USENIX Conference* (June 1987).
- [6] MIPS Computer Systems. *MIPS R4000 Microprocessor User's Manual*. 1991.
- [7] Blanck, G., and S. Krueger. "The SuperSPARC Microprocessor." *COMPCON* (February 1992): 136–141.
- [8] Dobberpuhl, Daniel, et al. "A 200 MHz 64b Dual-Issue CMOS Microprocessor." *Proceedings of the 39th International Solid-State Circuits Conference* (February 1992): 106–107.
- [9] Hewlett-Packard. *PA RISC 1.1 Architecture and Instruction Set Reference Manual*. 1990.
- [10] Intel Corporation. *Overview of the i860 XP Supercomputing Microprocessor*. 1991.
- [11] Chen, J. Bradley, Anita Borg, and Norman P. Jouppi. "A Simulation Based Study of TLB Performance." *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992): 114–123.
- [12] Wood, David A. and Randy H. Katz. "Supporting Reference and Dirty Bits in SPUR's Virtual Address Cache." *Proceedings of the 16th Annual International Symposium on Computer Architecture* (June 1989): 122–130.
- [13] Custer, Helen. *Inside Windows NT*. Microsoft Press, 1993.
- [14] Kessler, R. E., and Mark D. Hill. "Page Placement Algorithms for Large Real-Index Caches." *ACM Transactions on Computer Systems* 10, no. 4 (November 1992): 338–359.
- [15] Harty, Kieran, and David R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management." *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)* (October 1992): 187–197.
- [16] McVoy, L. W., and S. R. Kleiman. "Extent-like Performance from a UNIX File System." *Proceedings of Winter 1991 USENIX* (January 1991).
- [17] Babaoglu, Ozalp, and William Joy. "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits." *Proceedings of the 8th Symposium on Operating Systems Principles* (1981): 78–86.
- [18] Denning, Peter J. "The Working Set Model for Program Behavior." *Communications of the ACM* 11 (May 1968): 323–333.
- [19] Babaoglu, Ozalp. "Virtual Storage Management in the Absence of Reference Bits." Ph.D. thesis, Computer Science Division, University of California, Berkeley, 1981.

© Copyright 1993 Sun Microsystems, Inc. The SMLI Technical Report Series is published by Sun Microsystems Laboratories, Inc.
Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCEngine, SPARCworks, and SPARCCompiler are licensed exclusively to Sun Microsystems, Inc. VMS is a registered trademark of Digital Equipment Corporation. Windows NT is a trademark of Microsoft Corporation. Connection Machine is a registered trademark of Thinking Machines Corporation. R4000 is a registered trademark of MIPS Computer Systems Inc. Sun OS is a trademark of Sun Microsystems, Inc. i860 is a trademark of Intel Corporation. PA-Risc is a trademark of Hewlett-Packard Company. CHALLENGE is a trademark of Silicon Graphics, Inc. CHORUS is a registered trademark of Chorus Systems. All other product names mentioned herein are the trademarks of their respective owners.