Using 4KB Page Size for Virtual Memory is Obsolete

P. Weisberg and Y. Wiseman
Department of Computer Science
Bar Ilan University
{weisberg,wiseman}@cs.biu.ac.il

Abstract

A 4KB page size has been used for Virtual Memory since the sixties. In fact, today, the most common page size is still 4KB. Choosing a page size is finding the middle ground between several factors. On the one hand, a smaller page will reduce fragmentation; thus saving memory space. On the other hand, a larger page will increase TLB coverage; thus eliminating the need to access memory resident page tables. During the years that the 4KB page has been employed, memory size has increased from Megabytes to Gigabytes. We can sacrifice some space for higher performance.

With the aim of obtaining the optimal page size, we simulated applications from the SPEC2000 suite. We measured TLB misses and memory usage of all page sizes that are powers of two, ranging from 4KB to 256KB. Based on our results, we show that the use of 16KB size for the base page is the recommended selection.

Another way of increasing page size and TLB coverage is using superpages. Many machines support several page sizes, which let the OS use several page sizes. In this paper we survey various ways of making the most of superpages. We adopt a simple solution of superpaging with two page sizes. Based on our results, we suggest a base page of 16KB for code or small data segments and a larger page of 256KB for large data segments.

Keywords: Allocation/deallocation strategies, Simulation, Main memory, Virtual memory.

1. Introduction

Choosing the best page size for Virtual Memory requires considering several factors. A smaller page size reduces the amount of internal fragmentation. On the other hand, a larger page needs smaller page tables. The Linux kernel represents every physical page with a data structure which consumes about 1% of memory assuming 4KB pages. Enlarging the page size will reduce this table accordingly. The time required to transfer a page from or to disk is composed of access and transfer times, access time being the dominant factor. A larger page minimizes I/O time.

However, the main reason to prefer a larger page is to increase the virtual to physical translation speed. In a virtual memory system, every address issued by the CPU is translated into a physical address by the memory management unit (MMU) hardware. The translations are stored in memory resident page tables. To reduce address translation time, the most recently used translations are kept in a Translation Lookaside Buffer (TLB), a fast access cache. In case of a TLB miss, the translation must be searched in the page table and loaded into the TLB. Because access to a TLB must be fast, a TLB is made with a small number of entries. TLB coverage is the amount of memory mapped by the TLB. Assuming a TLB size of 100 entries and a page size of 4KB, TLB coverage is less than half a megabyte of memory. Since the size of a TLB is limited, in order to increase TLB coverage we have to use larger pages.

A 4KB page size has been used for Virtual Memory since the sixties. Even the PDP-11 had 8KB pages which are larger than the 4KB pages of contemporary IA-32. In fact, today, the most common page size is still 4KB. However, during the years the balance between the factors influencing page size has changed. Memory size of computers has increased from Megabytes to Gigabytes, as a result only a small fraction of the memory of contemporary computers is covered by TLB. Access times of disks have not kept up with throughput increases. In recent years, throughput has improved by a factor of 100, but access time has improved by only a factor of 3 [9]. Transferring larger pages from and to disk became more efficient.

2. Superpaging

Another possibility of increasing TLB coverage is by using superpages. Many modern architectures support superpages, which let the OS use several page sizes. Superpaging enables choosing an appropriate size for an allocation – a small page for a small spatial locality to save memory, and a large page for a large locality to increase TLB coverage.

Pages larger than 1MB are suitable for the allocation of non-paged memory, such as for mapping frame buffers or for fixed parts of the kernel. When using superpages for paging the code and data of user programs, a medium sized page should be preferred. The cost of internal fragmentation with very large pages may be too high. Writing to such a page is also a problem, since there is only one dirty bit and there is a need to update the whole page.

Superpages can be used by the operating system in three different ways:

2.1. Allocation

At page fault time a large page is allocated and all its subpages are loaded into the memory. The page is initially mapped in the page table as a large page. Commercial OSes like Solaris MPSS [3] use this method for superpaging.

Implementations of multiple pagesizes on IRIX [1] and HP-UX [6], also allocate large pages at fault-service time. In IRIX the desired page size is specified by the user prior to running the application or via a system call compiled into the code. In HP-UX the size of the page is specified either by the user or determined transparently by the operating system according to the size of the memory region required.

Allocating and populating a large page at page fault time has the advantage that any reference to this page from the moment of allocation does not incur a TLB miss and uses a single TLB entry. Transferring a whole large page from secondary storage is more efficient than transferring the small pages separately.

2.2. Reservation

In a reservation-based allocation, upon first referencing an address in a superpage, the superpage is allocated but not populated. At first, only the base pages causing page faults are loaded from secondary storage to memory. When the number of populated frames reaches a threshold, the missing pages are brought from disk and the small pages are promoted into a superpage.

Talluri et al. [8] propose a reservation scheme for use with a TLB that can map two fixed page sizes; the suggested sizes are 4KB and 64KB. Navarro [4] proposes a reservation scheme with multiple page sizes in which promotion can be performed incrementally.

The advantage of reservation as compared to full allocation is that it postpones loading of base pages until it is more certain that the subpages will be used as a superpage. It also makes the start-up time of a process shorter.

2.3. Relocation

In a relocation based scheme, base pages are faulted in regularly and memory usage is monitored to decide when it is worthwhile to create a superpage. In that case a contiguous area of memory is found and the base pages are copied. If not all pages are resident the missing pages are brought from secondary storage.

In another paper, Talluri et al. [7] propose a relocation scheme in which there are two page sizes 4KB and 32KB. The threshold they use for relocation is whether at least half or more of the base pages have been accessed. Romer et al. [5] propose a scheme for tracking potential superpage usage and deciding dynamically when to promote the base pages. According to this approach superpages are created only when necessary thus minimizing internal fragmentation.

3. Choosing a page size

To find the desired size for paging in virtual memory, we ran applications from the SPEC2000 suite. We counted the TLB misses and noted the memory usage of various page sizes. A right choice for a page size is when there is a significant decrease in TLB misses without a significant increase in memory usage.

When using superpaging, operating systems such as HP, IRIX and Solaris use the allocation method. They allocate a large page at page fault time. This is due to the simplicity of allocating and mapping the whole superpage when first referencing it and because the benefit of other more complex methods is not clear. We would like to adopt this scheme.

Many machines which offer superpages have a variety of page sizes. However, there is no clear policy for the OS to select the right size. So we would like to adopt a simplified superpaging system in which just one of two page sizes is chosen, either a base page or a large page. According to this scheme, the policy of the operating system is to allocate a fixed large page if the memory object is large enough and if there is enough memory. While running the SPEC2000 suite we noticed that dTLB misses drop only with larger pages. This may help us figure out the large page size suitable for large data segments.

4. Experimental Setup

To measure the TLB misses and memory usage of various page sizes, we simulated 12 applications from the SPEC2000 suite. Simulation of these benchmarks to completion takes a long time, so we traced each application for 48 hours. The computer used was a 3.66GHz Intel(R) Xeon(TM) CPU; it was dedicated to this simulation. We used `valgrind` - a suite of simulation-based debugging and profiling tools for Linux. One of its tools `Lackey` was adapted to output a trace of memory references. The output consisted a trace of page references for all power of 2 multiples from 4KB up to 256KB.

The trace creates immense amounts of output. If the output is saved to a file, it will grow quickly to gigabytes. To overcome this limitation, we used on-the-fly simulation. The traced output of `valgrind` was input via a pipe to a program which analyzed the data and produced the results.

The analyzing program simulated an LRU based TLB and counted the TLB misses for each of the page sizes. The simulated TLB was a fully associative TLB having 64 entries for instructions and 64 entries for data. We assumed the TLB is dedicated to one application, and did not consider operating system or other applications that may occupy slots in the TLB. As was mentioned earlier, the operating system can use big pages and thus occupy a small space in the TLB. In that case there is minor influence of a running operating system on TLB performance.

In the analyzing program, we also counted the pages allocated to each application. We calculated for each page size the overall amount of memory the program occupied during its execution. We assumed a large amount of memory with no need to do swapping.

5. Results

Fig. 1 illustrates the TLB misses and memory usage of four applications with increasing page sizes. The instructions behavior is shown on the left four graphs and that of data on the right four graphs. In each graph each page size has two columns; the left column presents the absolute number of TLB misses that occured while running the application and the right column the total amount of memory used.

In the the graphs of `crafty` and `parser`, the iTLB misses drop very low when using 16KB pages. In fact, these two applications represent 10 out of the 12 applications that we ran. The relative iTLB miss percentage is close to 0% for most applications when using 16KB pages. As to memory usage, We can also see that the relative increase in memory usage for `crafty` and `parser` is less than 25%. The extra memory that is consumed seems to be worth the performance boost. Even for the applications `apsi` and `vpr` iTlb decreases to about 1/3 and memory increases by about 2/3 at 16KB, which may be acceptable considering today's computers which have plenty of memory.

It can be concluded that a 16KB base page is the right size for the allocation of code; it almost eliminates iTLB misses for most applications without incurring a high memory cost. Allocating larger pages is not worthwhile since for most applications it hardly improves iTLB misses and for some applications it considerably increases memory usage.

Looking at the data of `crafty` and `parser` we see that dTLB misses drop very low only when using 256KB pages. This is the behavior of most applications, however the dTLB misses of `vpr` (and `gcc`) get very low at 32KB pages and that of `apsi` not even at 256KB. As to memory usage for data, it is almost constant for increasing page sizes.

It can be concluded, that for large data objects it is worth allocating 256KB pages. However, for small data objects 16KB pages should be allocated, the same size as the base page used for code.

References

[1] N. Ganapathy and C. Schimmel. General Purpose Operating System Support for Multiple Page Sizes. In Proceedings of the USENIX 1998 Annual Technical Conference. USENIX Assoc., June 1998.

[2] Gokul B. Kandiraju, and Anand Sivasubramaniam. Characterizing the dTLB Behavior of SPEC CPU2000 Benchmarks. ACM SIGMETRICS Perform. Eval. Rev., Vol. 30, No. 1, pages 129–139, 2002.

[3] Richard McDougall. Supporting Multiple Page Sizes in the Solaris Operating System. Sun BluePrints OnLine, March 2004.

[4] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI), December 2002.

[5] Ted Romer, Wayne Ohlrich, Anna Karlin, and Brian Bershad. Reducing TLB and memory overhead using online superpage promotion. In Proc. of the 22nd Annual Int. Symp. on Computer Architecture, pages 176–187, June 1995.

[6] Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of Multiple Pagesize Support in HP-UX. In Proceedings of the USENIX 1998 Annual Technical Conference, June 1998.

[7] M.Talluri, Kong, S., Hill, M., and Patterson, D. Tradeoffs in Supporting Two Page Sizes. In Proceedings of the 19th Annual International Symposium on Computer Architecture, pages 415–424, May 1992.

[8] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In Proc. 6th ASPLOS, pages 171–182, October 1994.

[9] Tom's Hardware. 15 Years Of Hard Drive History: Capacities Outran Performance. http://www.tomshardware.com/reviews/15-years-of-hard-drive-history,1368.html.

[10] Rafael B. Yehezkael, Yair Wiseman, Haim G. and Mendelbaum and I. L. Gordin, Experiments in Separating Computational Algorithm from Program Distribution and Communication, LNCS of Springer Verlag Vol. 1947, pp. 268-278, 2001.

[11] Yair Wiseman, A Pipeline Chip for Quasi Arithmetic Coding, IEICE Journal - Trans. Fundamentals, Tokyo, Japan, Vol. E84-A No.4, pp.

1034-1041, 2001.

[12] Yair Wiseman and Erick Fredj, Contour Extraction of Compressed JPEG Images, ACM - Journal of Graphic Tools, Vol. 6(3), pp. 37-43, 2001

[13] Shmuel T. Klein and Yair Wiseman, Parallel Huffman Decoding with Applications to JPEG Files, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 46(5), pp. 487-497, 2003.

[14] Yair Wiseman and Dror G. Feitelson, Paired Gang Scheduling, IEEE Transactions on Parallel and Distributed Systems, Vol. 14(6), pp. 581-592, 2003.

[15] Mordechay Geva and Yair Wiseman, A Common Framework for Inter-Process Communication in a Cluster, Operating Systems Review, Vol. 38(4), pp. 33-44, 2004.

[16] Yair Wiseman, Karsten Schwan and Patrick Widener, Efficient End to End Data Exchange Using Configurable Compression, Proc. The 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, pp. 228-235, 2004.

[17] Maayan Geffet, Yair Wiseman and Dror G. Feitelson, Automatic Alphabeth Recognition, Kluwer Journal of Information Retrieval, Vol. 8(1), pp. 25-40, 2005.

[18] Shmuel T. Klein and Yair Wiseman, Parallel Lempel Ziv Coding, Journal of Discrete Applied Mathematics, Vol. 146(2), pp. 180-191, 2005.

[19] Yair Wiseman, ARC Based SuperPaging, Operating Systems Review, Vol. 39(2), pp. 74-78, 2005.

[20] Yair Wiseman, Advanced Non-Distributed Operating Systems Course, ACM - Computer Science Education, Vol. 37(2), pp. 65-69, 2005.

[21] Moses Reuven and Yair Wiseman, Reducing the Thrashing Effect Using Bin Packing, Proc. IASTED Modeling, Simulation, and Optimization Conference, MSO-2005, Oranjestad, Aruba, pp. 5-10, 2005.

[22] Moses Reuven and Yair Wiseman, Medium-Term Scheduler as a Solution for the Thrashing Effect, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 49(3), pp. 297-309, 2006.

[23] Yair Wiseman, The Relative Efficiency of LZW and LZSS, Data Science Journal, Vol. 6, pp. 1-6, 2007.

[24] Yair Wiseman and Irit Gefner, Conjugation Based Compression for Hebrew Texts ACM Transactions on Asian Language Information Processing, Vol .6(1), article no. 4, 2007.

[25] Mordechay Geva and Yair Wiseman, Distributed Shared Memory Integration, Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2007), Las Vegas, Nevada, pp. 146-151, 2007.

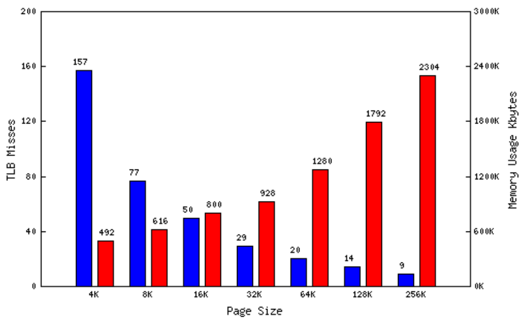
[26] Yair Wiseman, Burrows-Wheeler Based JPEG, Data Science Journal, Vol. 6, pp. 19-27, 2007.

[27] Ilan Grinberg and Yair Wiseman, Scalable Parallel Collision Detection Simulation, Proc. Signal and Image Processing (SIP-2007), Honolulu, Hawaii, pp. 380-385, 2007.

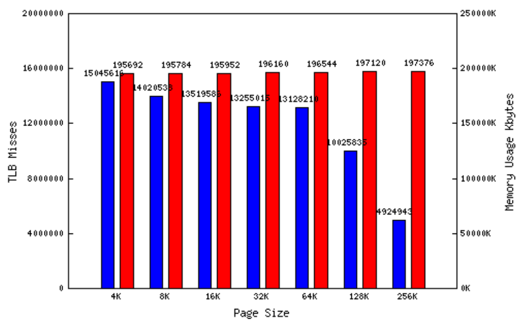
[28] Yair Wiseman, ASOSI: Asymmetric Operating System Infrastructure, Proc. 21st Conference on Parallel and Distributed Computing and Communication Systems, (PDCCS 2008), New Orleans, Louisiana, pp. 193-198, 2008.

[29] Yair Wiseman, Joel Isaacson and Eliad Lubovsky, Eliminating the Threat of Kernel Overflows, Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008), Las Vegas, Nevada, pp. 116-121, 2008.

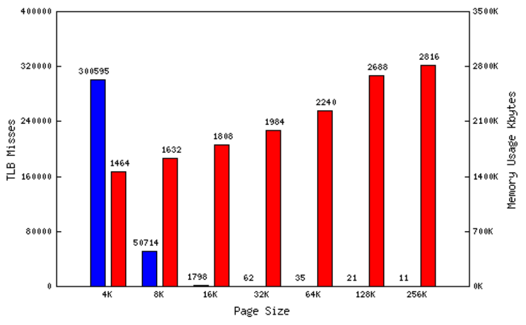
[30] Moshe Itshak and Yair Wiseman, AMSQM: Adaptive Multiple SuperPage Queue Management, Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008), Las Vegas, Nevada, pp. 52-57, 2008.



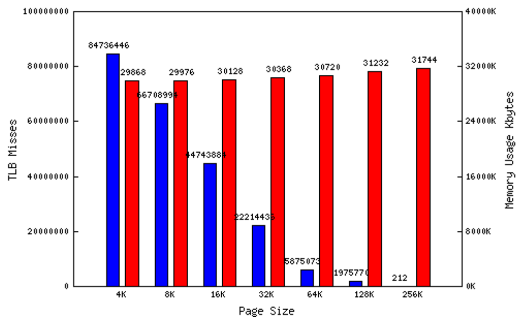
(a) apsi instructions



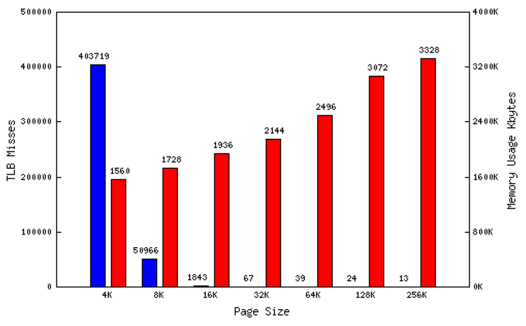
(b) apsi data



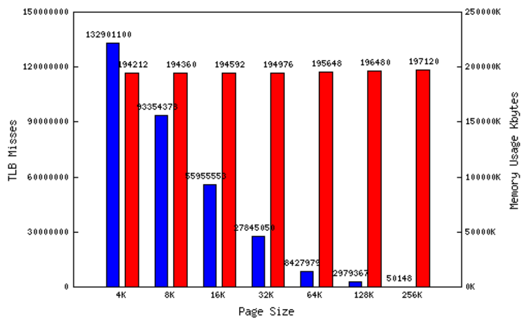
(c) crafty instructions



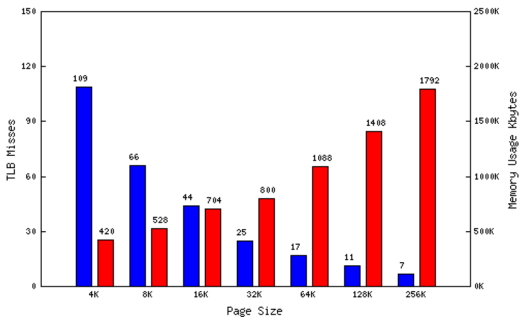
(d) crafty data



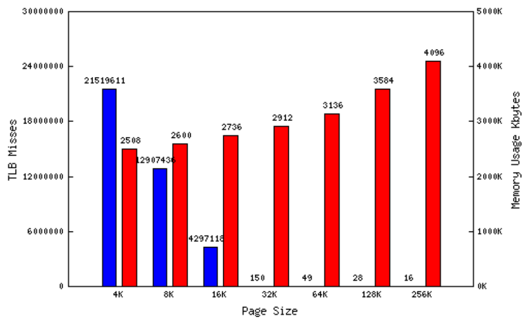
(e) parser instructions



(f) parser data



(g) vpr instructions



(h) vpr data

Figure 1. TLB misses (blue bars) and Memory usage (red bars) for page sizes 4KB – 256KB. On the left side is the behavior of instructions and on the right side the behavior of data.