Preliminary Investigation of the RAMpage Memory Hierarchy

Philip Machanick[*] Pierre Salverda [†]
Department of Computer Science
University of the Witwatersrand
Private Bag 3
2050 Wits
South Africa

Abstract

The RAMpage memory hierarchy addresses the growing concern about the memory wall – the possibility that the CPU-DRAM speed gap will ultimately limit the benefits of rapid improvement in CPU speed. Reducing references to DRAM is an increasingly desirable goal as CPU speed improves relative to DRAM. As the cost of a DRAM reference increases, it makes increasing sense to consider options like pinning crucial parts of the operating system in at least the lowest-level cache, and to consider possibilities like context switches on references to DRAM. All these factors combine to make it increasingly desirable to treat DRAM as a paging device, while moving the main memory a level up to the lowest level of SRAM. The RAMpage hierarchy relegates DRAM to the role of a first-level paging device. Results presented here are for a preliminary simulation of the RAMpage hierarchy, and show that, if current memory system and CPU trends continue, the RAMpage strategy will become increasingly viable. Even with current miss costs and without implementing all features favourable to the RAMpage hierarchy, simulations show that it is possible to achieve run times up to 25% faster than those for a conventional hierarchy. Furthermore, RAMpage scales up better than the conventional hierarchy (as simulated) in that performance degrades less as DRAM reference costs increase.

keywords: memory hierarchy, caches, paging, software-controlled replacement

CR categories: B.3, C.4, D.4.2

1 Introduction

There has been much discussion in recent years of the “memory wall” [WM95, Joh95, Wil95] – a consequence of a growing CPU-DRAM speed gap [HJ91, BD94].

One approach to dealing with this growing CPU-DRAM speed gap is to focus on strategies for reducing misses, even if those strategies make each miss cost more. For example, work on software-controlled caches in the past [CSB86] may be more revelant today than when it was first done, given the increased cost of misses. However, even strategies for reducing misses run up against the problem that miss costs in thousands of instructions are hard to amortize even with very low miss rates.

[*] *<philip@cs.wits.ac.za>*
[†] *<psalverd@cs.wits.ac.za>*

While current CPU and DRAM speeds may not yet be in the league where DRAM should be considered a slow peripheral, it is worth considering the possibility that current trends will continue long enough that miss costs to DRAM will eventually be in the range of thousands of instructions.

If current trends persist, it starts to become worth considering options such as taking a context switch on a cache miss. Further, as misses become more expensive, it becomes worth exploring techniques for locking critical data or code in the lowest-level cache.

Taken to their logical conclusion, strategies like software-controlled caches, locking given data or code in the lowest level of cache and context switches on misses imply managing the lowest-level cache as a paged memory.

The approach proposed in this paper, the *RAMpage hierarchy*, is one in which the lowest-level cache (typically made of static RAM, SRAM) is instead considered to be the main memory, and DRAM is considered to be a first-level (L1) paging device. In the RAMpage model, disk becomes a second-level (L2) paging device.

The RAMpage hierarchy differs from the conventional hierarchy in the way specific levels of the hierarchy are managed; the total amount of hardware is not necessarily significantly different. It is possible that improvements to the basic RAMpage design will require additional hardware, but the same is true of enhancements to a basic cache architecture.

The RAMpage hierarchy will require some modifications to the operating system, since an additional level of paging is introduced. Given the significantly lower miss costs from SRAM to DRAM, as opposed to from DRAM to disk, the optimal page size at the new level will be smaller than traditional page sizes. Also, the faster access time to DRAM as compared with disk implies that trade-offs such as a more complex replacement algorithm versus more misses will differ in the SRAM-DRAM level as compared with the DRAM-disk level.

On the positive side, then, the RAMpage approach offers the potential to address the memory wall without extra hardware cost. The downside is that changes to the operating system are required (i.e., adopting a RAMpage architecture cannot be done simply by changing the hardware). Fortunately, these changes are relatively minor – the paging implementation can follow standard principles with minor differences to take into account the parameters of faults from SRAM to DRAM.

Figure 1 illustrates the traditional versus the RAMpage hierarchy. The major components (as illustrated here) are the same; detailed implementation differs.

In this paper, a preliminary investigation of the RAMpage hierarchy, in which measurements are made using trace-driven simulation, is presented. Several simplifications are made to reduce the number

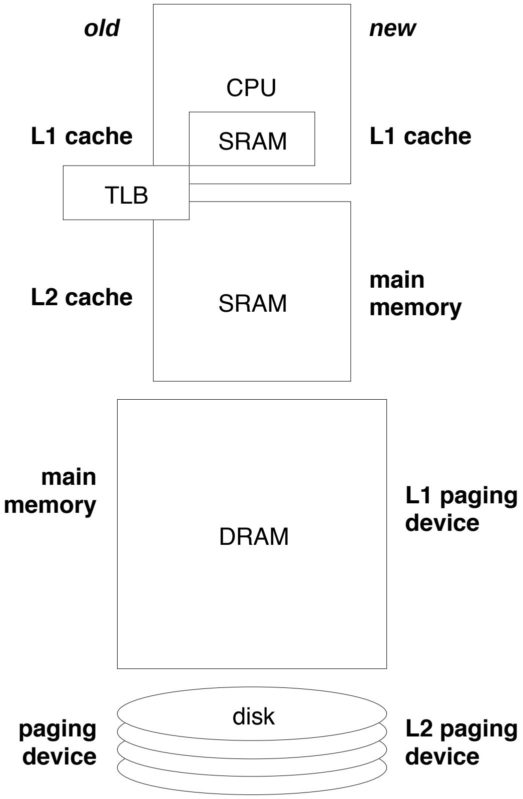

Figure 1: A Standard Hierarchy *vs.* a RAMpage Hierarchy

of variables in comparing the RAMpage hierarchy to a conventional two-level cache hierarchy. For example, context switches are not accurately modelled, nor are context swicthes taken on misses. However, a mix of programs is measured, to give an indication of how working sets of a multiprogramming mix develop over time. Also, a trace of a simple page replacement algorithm is used to simulate the costs of replacement in the RAMpage case.

The major aim of the paper is to show that, with the kinds of miss costs that are becoming common today, the RAMpage approach is viable even without pushing it to maximum advantage.

Preliminary findings are promising – taking into account that more attention needs to be paid to detailed simulation. If current trends continue, the RAMpage approach appears to be viable now, even without special hardware support. With extra investment in areas like hardware-supported page tables, the RAMpage approach could be even more attractive.

The remainder of this paper expands on issues raised so far, and presents results of measurement. The next section provides more background on trends, standard memory hierarchies in use today, and the implications of trends on such standard cache-based hierarchies. The following section briefly summarises related work, including software-managed caches and attempts at reducing the impact of the CPU-DRAM gap. After that is a more detailed description of the RAMpage architecture, followed by a section describing the experimental methodology and results. The final section presents conclusions.

2 Background

2.1 Introduction

Since the mid-1980s, when load-store (RISC) microprocessors became common, the trend in CPU speed improvement has been 50% to 100% per year, while DRAM speed improvement has only been 7% per year [HP95], leading to a doubling of DRAM reference costs, relative to CPU speed, every 6.2 years [BD94].

While there have been many attempts at reducing or hiding the costs of DRAM references [SF91, CB92, Jou90, BD94], if current trends continue, it will not be long before miss costs of thousands of instructions are commonplace. For example, it is possible to buy a DEC Alpha system today running at 500MHz with a peak instruction issue rate of 4 instructions per cycle, or an instruction per 0.5ns. At the same time, a typical cache miss cost today is of the order of 200ns. While miss costs are not an exact function of the CPU-DRAM speed gap (improved DRAM organization and bus design for example can reduce miss costs), the trend in the CPU-DRAM speed gap indicates that miss costs of thousands of instructions can be expected in the near future.

The remainder of this section presents a brief overview of standard cache-based hierarchies with virtual memory, and examines implications of the growing CPU-DRAM speed gap.

2.2 Standard Hierarchy

To put the research presented in this paper into perspective, here is a brief overview of common features of current architectures [HP95]. No attempt is made at covering all possible variations; the focus here is on issues to take into account when comparing RAMpage with a conventional hierarchy.

Typically, systems have one or more levels of cache – most recent designs have two, but three levels are likely to become commonplace as the CPU-DRAM speed gap widens. A cache is organized into blocks (also called lines). A cache reference which hits in the cache essentially divides into three phases:

- *indexing* – determining where in the cache to look
- *tag check* – determining whether the referenced block is present (and possibly updating its status)
- *selection within block* – do the read or write on the portion of the block referenced by the CPU

The CPU issues virtual addresses, which are translated somewhere during the cache reference process. A TLB (translation lookaside buffer) caches page translations, and is used where possible to do page translation within the cache reference cycle. In recent years, architectures have tended to use the virtual address on an index operation, and the physical address in the tags [KH92a], to allow more time for the address translation, though few have gone as far as to use completely virtually addressed caches [CSB86].

There are many other details of caches which could be considered [Smi82, PHH88]. Of most interest here are replacement strategies in caches, and the role of the TLB.

Caches are generally organized with a varying degree of associativity. No reasonably-sized cache is *fully associative*[1]: the situation where any block can be placed anywhere in the cache. Speed and cost constraints drive cache designers towards a *direct-mapped* cache, where a given block is always found in the same place. A compromise is an *n-way associative cache*. If cache size is m, and $1 \leq n \leq m$, there are n possible locations for placement of a given block, in an n-way associative cache. The associativity of a cache impacts on the replacement strategy. If the cache is direct-mapped, replacement is trivial. The higher the associativity, the more useful it is to have a more sophisticated approach to replacement, as there are more choices of blocks *not* to replace.

TLB misses can account for a significant portion of execution time. Some studies have shown that operating system code in particular accounts for a large portion of TLB miss costs [NUS $^+$93].

It has also been observed that code which uses large numbers of small data structures randomly scattered over memory can have a high number of TLB misses even if cache misses are low. Such a reference pattern is likely to be common in object-oriented code, especially where no special attention is paid to where objects are placed in memory [CGHM93].

In the best case, a TLB miss can be handled from a page table entry in L1 cache; in the worst case, a missing TLB entry may only be present on disk.

2.3 Implications of Trends

The growing CPU-DRAM speed gap has various implications for memory system designers.

One important implication is that misses should be reduced as far as possible. Although DRAM references are still much faster than disk references, some of the logic which applies to page faults starts to become an issue for DRAM references. For example, a more expensive replacement strategy in the lowest level of SRAM can be justified if it is more than offset by the reduction in misses. Also, it becomes useful to pin critical resources in the last SRAM level above DRAM.

The way a cache is organized, it is difficult to pin specific blocks into the cache, especially if there is a low degree of associativity. In the extreme case of a direct-mapped cache, pinning anything into the

[1] A small cache like a TLB is often fully associative.

cache means that any other block that maps to the same location can never be present in the cache.

While increasing associativity can address both miss reduction and making it possible to pin blocks in the cache, increasing associativity increases hit time [HP95]. Furthermore, pinning blocks in a cache would require either an extra tag bit (more hardware) or some other extra overhead on choosing a candidate for replacement.

Concerns of running into the memory wall need to take into account all aspects of the memory hierarchy which may access DRAM – not just the caches but also the TLB.

Page table entries likely to be needed soon but which cannot be accommodated in the TLB are an example of data which could profitably be pinned into the lowest SRAM level. Another example is code and data structures in the operating system for handling context switches.

Ultimately, the only way to deal with the memory wall is to find other work to do on a miss – in other words, to move towards context switches on misses.

Taking all these points into account, there is a strong case for investigating the RAMpage strategy.

3 Related Work

3.1 Introduction

The possiblility that the growing CPU-DRAM speed gap will eventually result in computer systems running into the memory wall makes it useful to re-evaluate past approaches which may still be applicable, or which may now be more applicable than when they were first investigated.

The remainder of this section briefly summarizes previous work on software-managed caches, and goes on to consider a range of approaches to reducing the impact of the CPU-DRAM speed gap.

3.2 Software-Managed Caches

In the 1980s, there was some work on software-managed caches [CSB86]. Software managed caches have the potential to reduce misses by more intelligent replacement than is feasible with hardware-managed caches. As with the RAMpage model, the trade-off is higher replacement costs.

In the 1980s, miss costs were not high enough to make software-managed caches viable. Also, CISC architectures in common use at the time had relatively high costs of traps to the kernel. More recent load-store designs, such as the MIPS architecture, make it possible to do a light-weight trap to the kernel, i.e., a trap does not automatically dump the entire machine state on the stack [KH92a].

The RAMpage model is potentially more viable than the previous approaches to software-managed caches for several reasons. It goes further towards treating the lowest level of SRAM as a fully software-managed level, i.e., a paged memory, with benefits outlined in Section 4. It also benefits from the increase in the CPU-DRAM speed gap since the work on software-managed caches, as well as the potential for relatively low-cost traps in modern CPUs.

3.3 Reducing the Impact of the CPU-DRAM Gap

There has been considerable attention directed at reducing the impact of the growing CPU-DRAM speed gap; only a short summary is presented here. Broadly, approaches can be divided into hardware and software strategies.

Hardware strategies can be divided into three major categories; a few examples are given here to illustrate the general principles:

- *reducing miss costs* – variations on DRAM organization including some cache on the DRAMchip, or approaches to capitilize on locality (fast page mode, EDO, etc.); more sophisticated buses; more heavily interleaved RAM
- *reducing number of misses* – more associativity, larger caches
- *hiding miss costs*– victim caches to hold recently replaced data [Jou90]; non-blocking caches [CB92] to support prefetch or continued out-of-order execution

Software strategies include compiler optimizations [BGS94], algorithm analysis which takes misses into account [LRW91], application restructuring [CGM91], and using page placement to reduce conflict misses [BLRC94, KH92b].

All of these strategies run up against one major fundamental problem: they are at best hiding the underlying trend, not changing it. At some point, a miss to DRAM has to be taken, and when it occurs, if current trends continue, the memory wall remains an issue. The RAMpage approach, with its potential for taking a context switch on a miss, offers the option of a detour around the memory wall.

4 The RAMpage Architecture

4.1 Introduction

Given the problems caused by the increasing CPU-DRAM speed gap, it is useful to attempt to find a solution that takes us further than previous approaches.

To summarize the issues raised in Subsection 2.3, it is possible to trade fewer misses for a higher replacement cost, and it would be useful if more operating system traffic could be kept to the SRAM levels of the hierarchy – including TLB misses and context switch code. By making it easier to manage what is at least in the lowest level of SRAM, it should become possible to avoid the worse cases of behaviour of TLB misses (at least in the case where a reference to DRAM or disk is not otherwise required).

Another point to consider is whether it is possible to arrive at a simpler strategy for hits, as compared with a TLB lookup and a cache reference. The result could be faster hits.

Finally, given the growing cost of DRAM references, it may be useful to have the option of taking a context switch on a miss to DRAM. This is not as fanciful as it may at first appear: early virtual memory system's page fault costs were not much different from current miss costs to DRAM[2].

All these issues lead to the idea of the RAMpage architecture – a revision of the conventional hierarchy in which main memory moves up a level to the lowest level of SRAM, while DRAM becomes the L1 paging device. Disk becomes the L2 paging device.

A major design issue here is to address the problem of high TLB miss costs. In particular, it is a concern that a hit in any SRAM level should not be slowed down by a slow TLB miss. While TLB miss costs may appear to be secondary to the other issues raised, there is little point in improving other aspects of the memory hierarchy, only to have TLB misses remain a bottleneck.

The remainder of this section provides more detail of the RAMpage architecture, followed by some implementation detail. Since this paper aims to investigate feasibility, detail is not developed as fully as if an implementation were currently proposed.

4.2 RAMpage Architecture Fundamentals

It is important to note that a basic RAMpage architecture need not cost more than a conventional cache-based architecture. The major components are the same. It is possible that additional hardware will be justifiable to improve performance, but the same is true of a cache: a simple, basic design can be enhanced by adding extra hardware, such as a victim cache to hold recently replaced blocks [Jou90]. In fact, the equivalent concept to a victim cache is commonly implemented in paged systems without extra hardware (recently replaced pages are kept on a *standby page list*; the page which was on the list longest is the one actually discarded [Cro97]).

To expand on Figure 1, the major difference at the hardware level is that the RAMpage architecture does not use cache tags in the lowest level of SRAM. However, whether there is a net increase or decrease in SRAM used for page tables as opposed to tags depends on the page size of the RAMpage system, an issue which is taken up in Section 5, where results are presented. The approach here is to use the same amount of SRAM in all simulations, to avoid having cost as a major issue.

[2] Astonishing though it may seem, in the 1950s, drum memories were being produced with sub-millisecond access times: see for example <http://www.cc.gatech.edu/gvu/people/Faculty/Randy.Carpenter/folklore/dec92-v1n2.html>.

At the software level, the major difference between RAMpage and conventional architectures is that an additional level of page table is needed.

Since the physical address space of the SRAM main memory is relatively small compared with the total address space, a sparse page table approach is needed. In this work, an inverted page table [HH93] is used to map the entire SRAM main memory to virtual addresses.

It is useful to ensure that the part of the page table needed for any hit to *any* SRAM level is in the SRAM main memory. Even if the inverted page table needs to do more work for a lookup than a forward page table, the worse cases of fetching a page table entry from DRAM or from disk will never occur for a reference which hits in one of the SRAM levels.

In the RAMpage model, a hit in any SRAM level, in the best case (in terms of TLB behaviour), proceeds from a TLB hit to a reference in either one of the the SRAM levels memory. In the worst case, a hit in an SRAM level requires that a TLB miss be handled from the lowest level of SRAM.

While a direct-mapped cache may be able to match the RAMpage architecture on hit cost, we argue that our organization is simpler and therefore easier to make fast in the best case, and the worst case is also better.

Once associativity is introduced into the cache, it becomes even harder to achieve a fast hit time. By comparison, the RAMpage approach is fully associative, which should reduce misses, yet is simpler in the case of a hit than even a direct-mapped cache. The cost is that replacements are more expensive than with a conventional cache architecture, as replacements are handled in software – in the style of a conventional paging implementation.

The RAMpage approach is slower on a TLB miss compared with a conventional architecture's best-case TLB miss, where the required page table entry is cached. In this case, the slower lookup of the RAMpage inverted page table results in lower performance for the RAMpage hierarchy.

The other areas where RAMpage will take a performance hit are in handling replacements and misses. Replacements are slower than with a cache since a software approach is used. Misses are slower since a page needs to be larger than typical cache blocks, as is illustrated in Section 5.

4.3 Preliminary Implementation Detail

Cache levels above the SRAM main memory are organized the same way in the RAMpage model as in conventional memory hierarchies.

The lowest SRAM level is organized like a conventional DRAM paged memory, except some changes are made to take into account the fact that the SRAM level is relatively small as compared to DRAM, and misses to DRAM are relatively quick as compared to page faults to disk. Also, some attention is paid to reducing TLB miss costs, especially where the reference concerned does not require access to DRAM or disk.

An inverted page table is used to implement page translation as, in this way, it can be guaranteed that any page present in the SRAM main memory is mapped in the portion of the page table present in SRAM main memory. A conventional forward-mapped table for a hierarchy where the main memory is relatively small would in any case have to be modified to take into account the relatively small portion of the total virtual address space resident in the main memory.

Compared with a cache, an SRAM main memory does not require a cache tag index operation, checking for a hit, or extracting the appropriate sub-block. Once a TLB translation has been performed, the appropriate reference can be performed directly. While cache indexing and TLB lookup can be performed concurrently in a virtually indexed cache, checking for a hit and subblock extraction are overheads in the conventional model which are avoided in the RAMpage model. For simplicity, the simulation implemented for this work does not model the speed difference between RAMpage and cache hits. However, the greater simplicity of the RAMpage model should aid the hardware designer in improving performance of hits.

RAMpage replacements are handled using a simple text book clock page replacement algorithm [Cro97, MOO87]. The philosophy is that a simple approach with minimal cost, at the expense of slightly more misses, is the right approach while miss costs are still relatively low. As miss costs to DRAM increase, more sophisticated replacement strategies will be justified.

For purposes of this work, the L2 paging device – the disk – is not implemented, as this level of the hierarchy does not differ from that of a conventional cache-based architecture.

It is possible – depending on the page size – that an additional level of page table may need more memory than cache tags. The trade offs are further explored in Section 5. It should be noted however that page table entries would be present anyway in the lowest level of cache, so the trade-off is hard to determine in general. In the conventional cache-based system, these page table entries would only be for paging in the DRAM and disk levels, whereas in the RAMpage model, page table entries for both levels of paging might be present in the SRAM main memory. However, most page translations in the RAMpage model are likely to be for pages resident in the SRAM main memory; the L2 page translations are not needed for example to service TLB misses, except in the case where there is also a miss from SRAM to DRAM or disk.

5 Experimental Evaluation

5.1 Introduction

Since there are many variables in a memory hierarchy which could confuse evaluation of the RAMpage alternative, the RAMpage implementation and a cache-based hierarchy used for comparison have been kept as simple as possible.

To simplify variation in the hierarchy, trace-based simulation is used. Traces are drawn from SPEC benchmarks[3]. Each trace is approximately 1-million references. A total of 20 traces is used, with a mix of floating point and integer code. 1-million traces is not much to simulate a realistic working set, so we produced longer traces with an approximation to trace stitching [AHH88], by using the same trace repeatedly. Since this technique is obviously of limited accuracy, we do not rely heavily on these longer traces to draw conclusions.

Traces from each benchmark are interleaved, with a fixed number of references from each before switching to another, to simulate a multiprogramming workload. We have made no attempt at simulating the cost of context switches (other than the impact on contents of each level of the memory hierarchy).

The remainder of this section details specifics of the architectures as implemented for measurement, followed by measurement and discussion of the results.

5.2 Simulated Architectures

The simulated CPU cycle time is 2ns (500MHz). For simplicity, we have not simulated a superscalar architecture, nor have we attempted to take into account pipeline interactions with the memory system (other than stalls for misses).

The TLB is identical in both memory hierarchies. It is software managed, fully associative and contains 64 virtual to physical address translations which are shared between code and data references. TLB hits require a single processor cycle, and misses are managed by software which retrieves the required translation from the page table.

Both hierarchies employ the same primary cache organization. The on-chip cache is physically addressed, split (16Kbytes each for the I- and D-caches) and direct mapped. A write back strategy with a write allocate miss policy is adopted. Block size is fixed at 32bytes, with a full block fetched on a miss. Dirty misses cause the entire block to be written back to the next level. Both the I- and D-caches cycle at the same rate as the CPU, with read hits taking one cycle and write hits requiring two. Misses incur processor stalls until the required block has been loaded.

[3] A selection of 20 traces from SPEC benchmarks produced by Mark Hill was used, from the FTP location <ftp://tracebase.nmsu.edu/pub/traces/uni/r3000/pdt/>.

The L2 cache in the standard hierarchy is external, physically addressed and unified. It is direct mapped and write back with a write allocate miss policy. Cache size is fixed at 4 Mbytes with block size varied from 128bytes. Like the primary cache, full blocks are fetched on a miss and dirty misses result in the entire block being written back to DRAM. Cycle time is set at 30ns, and so 1 secondary cache cycle = 15 CPU cycles. Read hits require a single secondary cache cycle, and write hits twice that amount. An inclusion property between L1 and L2 caches is maintained (i.e., the L1 cache is a subset of the L2 cache, other than that some L1 blocks may be dirty with respect to L2).

Main memory in the RAMpage model has the same *total capacity* as the secondary cache in the baseline system with 128byte blocks; this includes the space taken up by the L2 tags. It is assumed that each tag in the L2 cache requires 4bytes of storage, and so the 32K frames (each 128bytes) require a total of 32K $\times$ 4bytes = 128Kbytes of storage. Hence, the total capacity of the secondary cache is 4.125 Mbytes. Page size is varied in powers of two from 128bytes to 4096bytes. Inclusion is also maintained between the L2 cache and the SRAM main memory.

Adjacent layers of the hierarchies are connected by a 30ns bus which is 32bytes wide. Thus, transfer of a cache block between the L1 and the L2 cache (or SRAM main memory) requires a single bus cycle; transferring a block between L2 (or SRAM main memory) and DRAM requires requires four or more bus cycles (depending on block size).

An infinite DRAM is simulated, in which all data and code is preloaded, to avoid the need to simulate accesses to disk.

DRAM latencies are varied at 50 cycle increments from 65 cycles for reads and 60 cycles for writes, to 315 cycles for reads and 310 cycles for writes.

5.3 Results

In this subsection, results of trace-driven simulation are presented. These results are intended to provide a first cut at evaluating the RAMpage hierarchy, by comparison with a simple cache-based hierarchy.

Results are presented with a range of block (line or page) sizes in the lowest-level SRAM (L2-cache, or main memory, depending on the model being evaluated).

Table 1 contains simulated run times of both a conventional cache hierarchy and a RAMpage hierarchy. In each case block size at the L2 (or SRAM main memory) level is varied, and memory speed is varied. Run times are given in simulation time, rather than wall clock units.

It is interesting to observe that simulated times for the best block (or page) size for both hierarchies differ generally by about 1%. However, the block size for which the best performance occurs differs markedly between the RAMpage and cache-based architectures. The conventional architecture achieves

DRAM Speed	128bytes	256bytes	512bytes	1024bytes	2048bytes	4096bytes
rd: 65; wr: 60	71 101 033	72 079 551	74 144 325	77 871 338	85 153 856	100 005 393
	85 858 027	76 754 297	73 294 034	71 524 452	73 991 456	79 562 784
rd: 115; wr: 110	77 159 433	78 919 151	82 329 925	88 574 382	100 305 856	124 523 793
	90 714 428	82 032 697	79 154 034	78 260 452	83 565 856	92 253 984
rd: 165; wr: 160	83 262 569	85 823 916	90 454 725	99 093 738	115 457 856	148 830 528
	95 582 228	87 311 097	85 014 034	84 996 452	93 140 256	104 945 184
rd: 215; wr: 210	89 311 769	92 663 916	98 641 125	109 803 182	130 609 856	173 437 524
	100 450 028	92 589 497	90 874 034	91 732 452	102 714 656	117 636 384
rd: 265; wr: 260	95 298 633	99 503 916	106 827 525	120 417 582	145 761 856	197 911 124
	105 317 828	97 867 897	96 734 034	98 468 452	112 289 056	130 327 584
rd: 315; wr: 310	101 410 169	106 343 916	115 072 325	131 031 982	160 913 856	222 384 724
	110 185 628	103 146 297	102 594 034	105 204 452	121 863 456	143 018 784

Table 1: Hierarchy Performance. Total processor cycles after 20 million references, with varying DRAM speed and L2 page/block size, and context switches every 100000 references. Values for the standard hierarchy appear in the upper portion of each row, and those for the new hierarchy in the lower.

its best performance for the smallest block size used here, whereas RAMpage achieves the best performance for a 512byte page for slower miss costs, and a 1024 byte page for faster miss costs.

These numbers are to some extent an artifact of the simulation: no account has been taken of memory system enhancements which reduce the cost of subsequent references to DRAM, after the initial reference of a sequence of contiguous references. While all cases will be improved by a more realistic DRAM simulation, the RAMpage architecture will be improved more since it performs better on larger blocks (pages).

L2 page/block size	Standard Hierarchy		RAMpage Hierarchy	
	Reads	Writes	Reads	Writes
128bytes	106 052	14 396	97 356	0
256bytes	118 256	18 536	105 568	0
512bytes	138 640	25 072	117 200	0
1024bytes	173 920	38 304	133 984	0
2048bytes	238 592	64 448	168 128	23 360
4096bytes	369 536	120 832	204 416	49 408

Table 2: Total DRAM references (counted as number of bus transactions) incurred by the two hierarchies after simulation of 20-million references.

Table 2 illustrates how increasing the L2 block size or SRAM page size impacts on number of DRAM references. Of particular interest is that for the relatively small traces used here, there are no write backs at all in the RAMpage case, for smaller page sizes. Only with page sizes of 2K or more are there write backs, which confirms that the clock algorithm is more efficient in selecting appropriate candidates for write backs than is a simple direct-mapped cache.

The number of references here is not a direct indicator of performance, as even with our inefficient bus, multiple references forming a single miss or write back take less time than the equivalent number

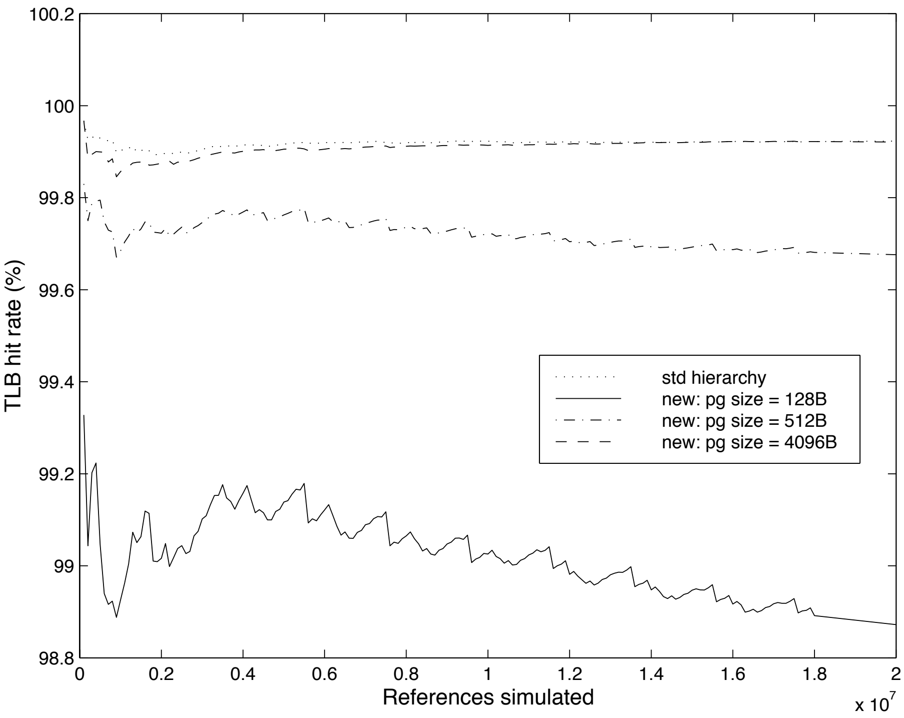

Figure 2: TLB Hit Rate. The TLB behaviour is the same for all block sizes of the conventional hierarchy.

of single references. Thus, the larger block or page sizes gain as long as increased number of references per miss is offset by the greater efficiency of handling multiple references for a given transaction.

A more significant factor as page size increases is the reduction in TLB misses, which is the largest factor in the improved performance of the RAMpage model for larger page sizes.

As can be seen in Figure 2, TLB hits increase significantly in the RAMpage case, as page size increases. However, TLB hits remain constant on the standard hierarchy as block size changes.

It should be noted that the TLB miss rates are not directly comparable across the two architectures. Of more interest is the actual TLB miss cost, since the RAMpage architecture is designed to accommodate all page translations within SRAM (provided the page concerned is in SRAM). Figure 2 illustrates how the new hierarchy performs badly with small pages. With small pages, there are many TLB misses. The data as given here shows TLB misses in terms of extra references generated by the TLB miss handler. A smaller fraction of the extra references for the RAMpage case go to DRAM, so the higher number of extra references generated by the TLB miss handler for page sizes of 512bytes and above is not significant.

Finally, it is of interest to see how performance develops for a longer trace.

Figure 4 illustrates how the new hierarchy starts to become a win once cold start misses are no longer an issue. This matches our view that the RAMpage architecture should come out ahead without too much work to optimize it. The data here is for the best cases of each hierarchy: 128byte blocks for the

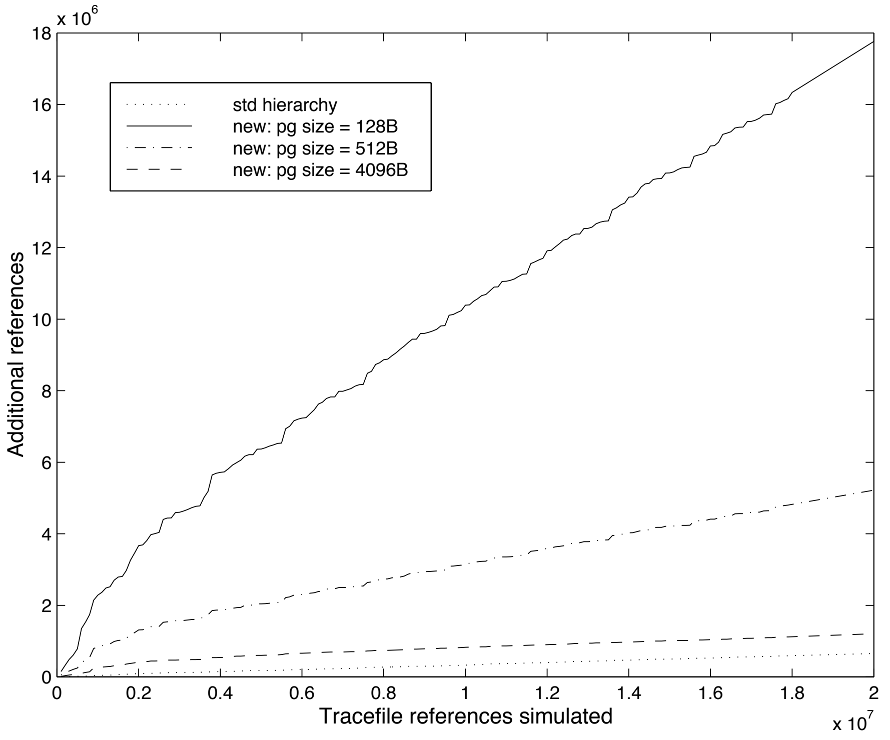

Figure 3: Cumulative TLB Miss Cost (in terms of extra instruction and data references). The TLB behaviour is the same for all block sizes of the conventional hierarchy.

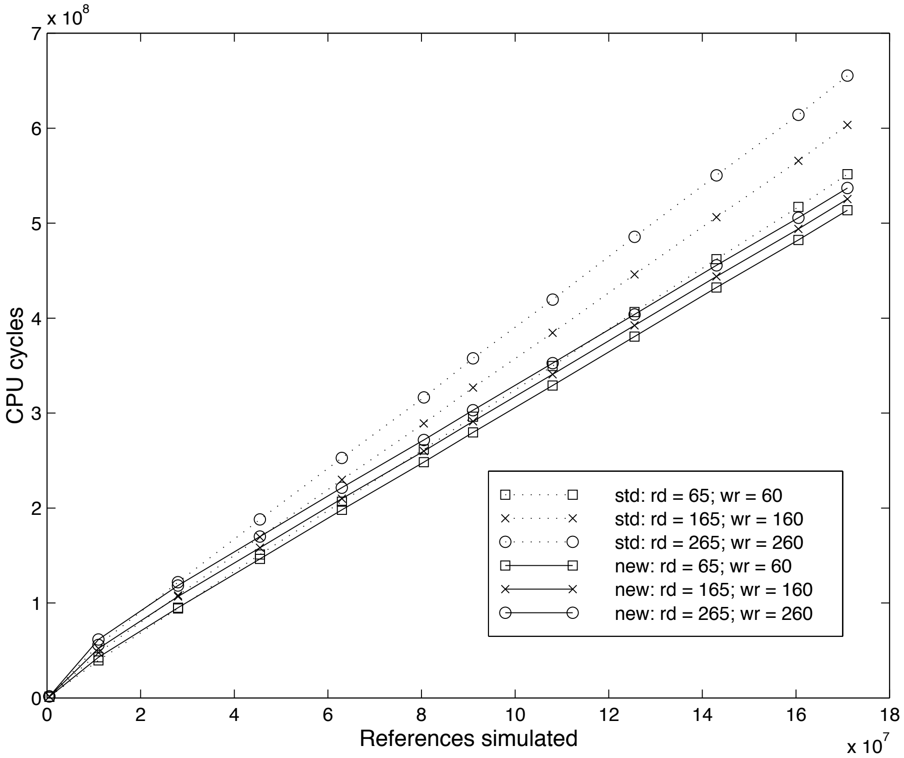

Figure 4: Cumulative Simulated Run Time (20-million references).

L2 cache, and 512bytes for the RAMpage model.

Two details are of particular interest. Even though the miss cost is increased by a factor of more than 4, the RAMpage hierarchy only slows down by about 5%. For the highest miss cost, the RAMpage hierarchy is about 25% faster than the conventional hierarchy.

However, since this 20-million reference trace was created by stitching together shorter traces, this data should be treated with some caution.

5.4 Discussion of Results

Despite the simplifications made to limit the variables in this initial work, the data presented here provides evidence that the RAMpage hierarchy is viable, especially as miss costs increase.

If the longer traces are typical of real runs, even with miss costs seen on some current machines, the simplified RAMpage architecture simulated here comes out ahead.

While TLB misses are a significant overhead in the new architecture, the simulation results show that, provided the page size is large enough, the extra number of TLB misses is not high enough to offset the gain in having most TLB misses serviced from SRAM. While large page sizes may cause extra overhead on misses, they also save overhead on page table size; fine-tuning details of this kind will require more detailed simulation, including a more realistic model of the memory system.

6 Conclusions

6.1 Introduction

The RAMpage work presented here is a beginning. Initial results indicate that it is worth further researching the RAMpage approach. This concluding section summarizes the major issues and results contained in this paper, then goes on to outline planned future work. In conclusion, the results are put into context.

6.2 Summary

The RAMpage memory hierarchy offers a way around the memory wall, by making it possible to do context switches on misses. This paper has shown that even without taking context switches on misses, the underlying infrastructure of the RAMpage approach has competitive performance with a simple conventional L2 cache-based system. Both the cache implementation and the RAMpage implementation could have many improvements made to them to make them more realistic, but the basic principle demonstrated in this paper is that the RAMpage approach is feasible.

6.3 Future Work

An important issue not addressed in this paper is that of context switches on misses. If miss costs are high enough (in this case, page faults from SRAM main memory to DRAM), it becomes worth while to take a context switch on a miss. Work is in progress to investigate the conditions in which it becomes worth taking a context switch on a miss to DRAM. Additions to the current simulation include more accurate simulation of context switch code and operating system data structures.

Another project under consideration is hardware support for inverted page tables. In this instance, the likely focus of the project will be on the use of formal methods to ensure correct implementation, since the design is likely to be relatively straightforward.

A possible benefit of the RAMpage approach is that object-oriented code will have fewer expensive TLB misses, given the RAMpage goal of ensuring that all pages currently found at least in part in any SRAM level have their translation at worst in the SRAM main memory. Users of languages such as Smalltalk or Java which use automatic memory allocation will particularly benefit from this improvement. Our future work will include investigation of the TLB behaviour of object-oriented code on a RAMpage versus a conventional cached machine.

Multiprocessor systems introduce another range of problems. To minimize variables in this initial research, multiprocessor systems are not considered. However, some lessons could be drawn from work on distributed shared memory (DSM) systems [BCZ90, DKCZ93].

6.4 Final Summary

Overall, the RAMpage approach appears to be promising. Further work is needed to investigate missing details. However, given initial data presented in this paper, there is reason to continue with further investigation.

The fact that an improvement of up to 25% was seen over the simple cache architecture is an indication that the RAMpage approach is viable, even without going to the extent of implementing currently missing details like context switches on misses. More important, the fact that RAMpage scales up better than the conventional cache-based architecture as miss costs grow, indicates that the RAMpage strategy will become more viable if current trends persist.

It somehow seems a more satisfactory metaphor to walk around a wall than to run into it.

Acknowledgements

Ian Sanders gave useful comments on this paper.

References

[AHH88] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[BCZ90] J.K. Bennet, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 125–134, Seattle, WA, May 1990.

[BD94] K. Boland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, August 1994.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high performance computing. *Computing Surveys*, 26(4):345–420, December 1994.

[BLRC94] B.N. Bershad, D. Lee, T.H. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

[CB92] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, September 1992.

[CGHM93] D.R. Cheriton, H.A. Goosen, H. Holbrook, and P. Machanick. Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: The value of distributed synchronization. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 159–162, San Diego, May 1993.

[CGM91] D.R. Cheriton, H.A. Goosen, and P Machanick. Restructuring a parallel simulation to improve cache behavior in a shared-memorymultiprocessor: A first experience. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 109–118, Tokyo, April 1991.

[Cro97] C. Crowley. *Operating Systems: A Design-Oriented Approach*. Irwin Publishing, 1997.

[CSB86] D.R. Cheriton, G. Slavenburg, and P. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 266–274, 1986.

[DKCZ93] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 144–155, San Diego, CA, May 1993.

[HH93] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 39–50, May 1993.

[HJ91] J.L. Hennessy and N.P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24(9):18–29, September 1991.

[HP95] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann Publishers, San Mateo, CA, 2nd edition, 1995.

[Joh95] E.E. Johnson. Graffiti on the memory wall. *Computer Architecture News*, 23(4):7–8, September 1995.

[Jou90] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[KH92a] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[KH92b] R.E. Kessler and M.D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, 1991.

[MOO87] M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft. *Operating Systems: Advanced Concepts*. Benjamin/Cummings Publishing, Menlo Park, CA, 1987.

[NUS$^+$93] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 27–38, May 1993.

[PHH88] S. Przybylski, M. Horowitz, and J. Hennessy. Design tradeoffs in cache design. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 290–298, May/June 1988.

[SF91] G.S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[Smi82] A.J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Wil95] M.V. Wilkes. The memory wall and the CMOS end-point. *Computer Architecture News*, 23(4):4–6, September 1995.

[WM95] W.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.