How Multithreading Addresses the Memory Wall

Philip Machanick
School of IT and Electrical Engineering, University of Queensland
Brisbane, QLD 4072
Australia
philip@itee.uq.edu.au

Abstract

The *memory wall* is the predicted situation where improvements to processor speed will be masked by the much slower improvement in dynamic random access (DRAM) memory speed. Since the prediction was made in 1995, considerable progress has been made in addressing the memory wall. There have been advances in DRAM organization, improved approaches to memory hierarchy have been proposed, integrating DRAM onto the processor chip has been investigated and alternative approaches to organizing the instruction stream have been researched. All of these approaches contribute to reducing the predicted memory wall effect; some can potentially be combined. This paper reviews several approaches with a view to assessing the most promising option. Given the growing CPU-DRAM speed gap, any strategy which finds alternative work while waiting for DRAM is likely to be a win.

Keywords: memory wall, CPU-DRAM speed gap, cache memories, DRAM, multithreading, chip multiprocessors, RAMpage

1 Introduction

This paper addresses the looming memory wall problem by proposing that strategies which find alternative work on a miss to dynamic random access memory (DRAM) will become increasingly important. With the launch of Hyper-Threading on the Pentium 4 in 2002, multithreaded architectures have become mainstream, so the contribution of multithreading and other alternative approaches to finding alternative work on a miss to DRAM is worth evaluating.

The memory wall is defined as a situation where the much faster improvement of processor speed as compared with DRAM speed will eventually result in processor speed improvements being masked by the relatively slow improvements to DRAM speed [Wulf and McKee 1995].

The problem arises from mismatches in *learning curves*. A learning curve is an exponential function of improvement with respect to time, arising from a constant percentage improvement per time unit. A well-known instance of a learning curve is Moore's Law, which predicts the doubling in the number of transistors for a given price every year.

As a consequence of Moore's Law, since the mid 1980s, processor speed has generally improved by 50-100% per year. DRAM speed on the other hand has only improved at about 7% per year, resulting in a doubling of the speed gap between processor and main memory cycle every 1 to 2 years.

The memory wall problem arises because the difference between two learning curves, each of which is an exponential function but with a different base, grows at an expontial rate. Specifically, if CPU speed improves by 50% in a year, a new model's speed can be expressed as a function of number of years, t, relative to an old model as $speed_{CPU_{new}}(t) = speed_{CPU_{old}} 1.5^t$; the learning curve for DRAM can be expressed $speed_{DRAM_{new}}(t) = speed_{DRAM_{old}} 1.07^t$. The speed gap grows at the rate of $(\frac{1.5}{1.07})^t \approx 1.4^t$, resulting in the gap doubling every 2.1 years.

Figure 1 illustrates the trends, assuming a conservative CPU speed improvement of 50% per year.

Technical Report December 2002. If accepted for publication, the final version will be found at `http://www.itee.uq.edu.au/~philip/Publications/`.

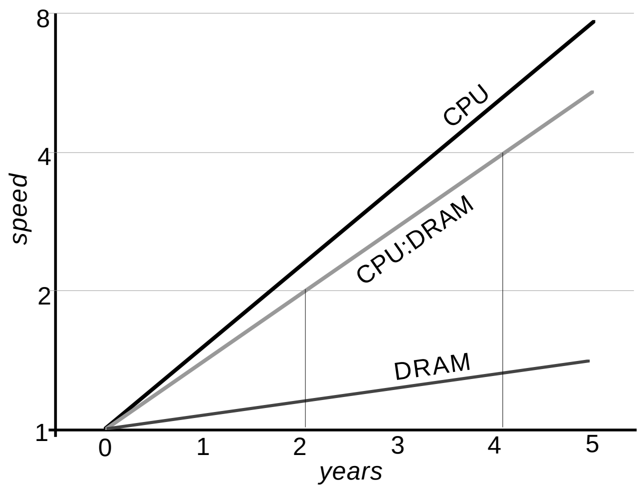


Figure 1: The growing CPU-DRAM speed gap, expressed as relative speed over time (log scale). *The vertical lines represent intervals at which the speed gap has doubled.*

Consequently, even if the number of memory references to DRAM is a relatively small fraction of the total, the fraction of overall run time spend waiting for DRAM – if the problem is not addressed – will grow over time, and eventually dominate the total time spent executing a program [Wulf and McKee 1995].

In the short term, the problem has been avoided by traditional techniques: making caches faster and reducing the miss rate from caches (by increasing the size or associativity, or both). However, as the processor-DRAM speed gap grows, relatively simple solutions are likely to be increasingly ineffectual.

By and large, attempts at addressing the problem have been piecemeal. Some have addressed the organization of DRAM, in an attempt at fending off the onset of the problem. Others have addressed cache organization. In addition, there have been investigations of options to keep the processor busy while waiting for DRAM.

This paper brings together a range of options which have been explored separately, and investigates linkages, as a basis for finding a more comprehensive solution, focused on the need to fill the increasing delay while waiting for DRAM.

The remainder of this paper is structured as follows.

Section 2 explains approaches at improving DRAM performance, especially those aimed at reducing or hiding latency. The approach here is to relate DRAM improvements specifically to the requirements of caches and processors.

Section 3 presents a range of improvements to caches which reduce the potential onset of the memory wall problem. Some of the ideas are relatively new; others are older ideas whose relevance has increased, given the growing latency of DRAM access (relative to processor speed).

Approaches to accommodating more than one thread of execution are compared in Section 4. This area presents a solution because it offers an option for keeping the processor busy while waiting for DRAM.

Finally, Section 5 wraps up the paper with a summary, discussion of the options, and some recommendations for future work.

2 Improvements to DRAM

2.1 Introduction

If DRAM is the problem, it is obvious that it is worth working on improvements to DRAM. However, if the underlying learning curve remains unchanged, such improvements are not going to alter the fundamental, underlying trend. This paper assumes that no big breakthrough is going to occur in DRAM design principles. As such, the improvements which have been made to DRAM bear the same relationship to the memory wall problem as improvements to other areas such as caches: they stave off the problem, rather than solving it.

Nonetheless, such improvements to DRAM are worth studying, for their potential interaction with other solutions.

The approaches examined here are attempts at reducing the latency of DRAM operations and, broadly speaking, ways of designing DRAM to the requirements of caches and the processor.

2.2 Reducing Latency

Although the assumption is made that the underlying learning curve is not changing, it is worth considering other sources of improvement. These improvements, by their nature will be once-off: after the improvement is made, the existing trend will continue, if from a slightly better base.

The most obvious improvement is to address off-chip delays, not only to reduce latency significantly, but also to address bandwidth limitations of off-chip interfaces [Cuppu and Jacob 2001], though the latter issue is less of a concern in terms of addressing the memory wall.

As the available component count on a chip grows, the range of design trade-offs increases. Increasingly complex processors have traditionally used much of the extra real estate, though on-chip L2 caches have become common. System overheads can account for 10–40% of DRAM performance, so moving DRAM on-chip has some attractions [Cuppu and Jacob 2001]. The IRAM project consquently proposes using future extra chip space instead to place a significant-sized DRAM on the processor chip – or, alternatively, adding a processor to the DRAM chip [Kozyrakis et al. 1997].

The IRAM project is motivated by the observation that off-chip latencies are at least one area where DRAM latency can be addressed, taking advantage of the large amount of chip real-estate made available as feature sizes shrink. Another good reason for the IRAM idea is that the classic use of extra chip real estate, increasing instruction-level parallelism, even if it delivers the intended speedup, places more stress on the memory system. Trading less peak throughput of the processor for a faster memory system becomes increasingly attractive as the CPU-DRAM speed gap grows.

Direct Rambus introduces options to pipeline multiple memory references. While the latency of a given operation is no lower than would be the case with other memory technologies if there is a single memory transaction, Direct Rambus lowers the effective latency of operations under load [Crisp 1997].

Direct Rambus is one of several designs aimed at improving overall characteristics of DRAM by placing more of the control on the DRAM chip. Other variations have included SRAM cache on the DRAM chip, to reduce the frequency of incurring the full DRAM access time; this idea has been taken up more recently in the DDR2 standard.

On the whole, though, latency reduction is limited to once-off improvements, which do not improve the underlying learning curve. Consequently, when all such improvements have been made, we are back at the situation where CPU speed is doubling relative to DRAM approximately every 1 to 2 years.

2.3 Relationship to Cache and Processor Requirements

Some DRAM improvements are a good match to cache requirements. Given that a typical cache block is bigger than the minimum unit transferred in one typical memory reference, burst modes such as those available in SDRAM and Direct Rambus, while failing to reduce the initial start-up latency, make it possible to move an entire cache block significantly faster than the time to do an equivalent number of random references.

Generally, sequential memory accesses, or reference streams with predictable behaviour, can be targeted for improvements in DRAM organization – including caches on the DRAM chip, and pipelining multiple references. However, completely random references must ultimately result in the processor seeing the full latency of a DRAM reference.

2.4 Summary

There have been many attempts at improving the organization of DRAM, a few of which have been surveyed here. All attempt to exploit such matches as can be found between the faster aspects of DRAM and the processor's requirements. Such improvements, however, have not changed the underlying trend. DRAM speed improvement remains much slower than processor speed improvement.

The threat of the memory wall remains as long as the underlying DRAM latency trend cannot be changed.

3 Improvements to Caches

3.1 Introduction

Seeking to improve caches is an obvious goal in the face of the growing processor-DRAM speed gap. While a cache does not address the speed of DRAM, it can hide it. Clearly, the goal should be to increase the effectiveness of caches at least as fast as the CPU-DRAM speed gap grows. This section addresses the extent to which cache improvements are keeping up.

Latency tolerance addresses the extent to which cache improvements are able to hide increases in DRAM latency.

If latency of DRAM cannot be improved, the number of times DRAM is referenced can potentially be reduced. Consequently, it is useful to examine techniques for reducing misses. These techniques include higher associativity, software management of caches and improvements in replacement strategy.

This section examines latency tolerance and miss reduction, followed by an overall summary.

3.2 Latency Tolerance

One approach to tolerating latency is to have other work for the processor on a stall; *simultanous multithreading* or *SMT* is such an approach [Tullsen et al. 1995]; SMT is examined in more detail in Section 4.

More specifically to cache design, some approaches to reducing the effect of latency include:

- *prefetch* – loading a cache block before it is requested, either by hardware or with compiler support [Mowry et al. 1992]
- *speculative load* – a load which need not be completed if it results in an error (such as an invalid address) but which may be needed later [Rogers and Li 1992]: a more aggressive form of prefetch – an idea taken up in the Intel IA64
- *critical word first* – the word containing the reference which caused the miss is fetched first, followed by the rest of the block [Handy 1998]
- *memory compression* – a smaller amount of information must be moved on a miss, in effect reducing the latency. A key requirement is that the compression and decompression overhead should be less than the time saved [Lee et al. 1999]
- *write buffering* – since a write to a lower level does not require that the level generating the write wait for it to complete, writes can be buffered. Provided that there is sufficient buffer space, a write to DRAM therefore need not incur the latency of a DRAM reference. There are many variations on write strategy when the write causes a miss, but the most effective generally include write buffering [Jouppi 1993]
- *non-blocking caches* – also called *lockup-free* caches: especially in the case where there is an aggressive pipeline implementation, there may be other instructions ready to run when a miss occurs, so at least some of the time while a miss is being handled, other instructions can be processed through the pipeline [Chen and Baer 1992]

These ideas have downsides. For example, any form of prefetch carries the risk of replacing required content from a cache sooner than necessary, resulting in unnecessary misses ("cache pollution" refers to this situation, especially where the prefetch turns out to be unnecessary). A nonblocking cache obviously requires an increase in hardware complexity – with accompanying difficulties in scaling speed up – to keep track of pending memory references which have not as yet been retired.

Some of these problems can be worked around. For example, prefetches can be buffered, avoiding the problem of replacing other content before they are needed, or replaced blocks can be cached in a victim cache [Jouppi 1990]. However, as with latency reduction techniques, the underlying latency of the DRAM is only being masked, and the learning curve remains the same.

The remaining approaches are of most use when the CPU-DRAM speed gap is not very great. Critical word first is little help if most of the time to do a memory reference is the initial setup, and the DRAM has a fast burst mode to read contiguous locations. Memory compression does not avoid the latency of a single memory reference, again the most significant factor to eliminate. Non-blocking caches work best when the level below that where the miss occurred has a relatively low latency, otherwise the extra time the CPU can be kept busy will not be significant. Only write buffering is likely to represent significant gains in the case of a relatively large speed gap, though the size of the buffer will need to be increased as the speed gap grows, to maximize the fraction of writes which can be captured.

Optimizing write performance, while useful, has limited value. Since all instruction fetches are reads, and a high fraction of data references are typically reads, the fraction of references which are writes is necessarily low. For instance, in one study, fewer than 10% of references in the benchmarks used were writes [Jouppi 1993].

Latency can also potentially be tolerated by finding other work for the processor, an issue explored further in Section 4.

In general, however, tolerating latency becomes increasingly difficult as the CPU-DRAM speed gap grows.

3.3 Miss Reduction

If tolerating latency is difficult, can the events when it must be tolerated be reduced? Clearly, reducing cache misses to DRAM is a useful goal. A few approaches to reducing the number of misses are explored here, to illustrate the potential for improvement.

First, increasing associativity in general reduces misses, by providing the opportunity to do replacement more accurately. In other words, conflict misses can be reduced.

In general, increasing associativity has speed penalties. Hit times are increased as the trade-off for reducing misses. Generally, high associativity in first-level caches is the exception, except with designs where lower-level caches are optional. For example, the PowerPC 750 has an 8-way associative L1 cache; the PowerPC 750 is part of a processor family which includes parts which cannot take an L2 cache (the 740 series).

As an example of design trade-offs which change over time, when it became possible for a limited-size L2 cache to be placed on-chip in the AMD Athlon series, a 256Kbyte unit with 16-way associativity was introduced. As cache size increases, the benefit from higher associativity decreases [Handy 1998], so it seem likely that such high associativity will not be maintained in future designs, where a larger on-chip L2 cache becomes possible.

If associativity has speed penalties, that creates an opportunity to investigate alternatives which achieve the same effect with different trade-offs, resulting in a higher overall speed. In some designs, hits are divided into cases, with the best case similar to a direct-mapped cache (simple, fast hits) and the worst case more like a set-associative cache (a higher probability of a hit, with some speed penalty). In others, alternative ways of achieving associativity have been explored which either have no extra hit penalty (with penalties in other areas), or which are justified as reducing misses and can therefore tolerate a higher hit time.

The RAMpage memory hierarchy achieves full associativity in the lowest-level cache by moving the main memory up a level. Main memory is then in SRAM, and DRAM becomes a first-level paging device, with disk a secondary paging device. RAMpage has no extra penalty for hits in its SRAM main memory because it uses a straightforward addressing mechanism; the penalty for fast hits is in software management of misses. RAMpage has been show to scale better as the CPU-DRAM speed gap grows than conventional hierarchies, but mainly because it can do extra work on a miss [Machanick 2000], so further discussion of RAMpage is found in Section 4.

Full associativity can be achieved in hardware without the overheads for hits associated with a conventional fully-associative cache, in an indirect index cache (IIC), by what amounts to a hardware implementation of the page table lookup aspect of the RAMpage model. An inverted page table is in effect implemented in hardware, to allow a block to be placed anywhere in a memory organized as a direct-mapped cache [Hallnor and Reinhardt 2000]. The drawback of this approach is that all references incur some overhead of an extra level of indirection. The advantage is that the operating system need not be invoked to handle the equivalent of a TLB miss in the RAMpage model.

As cache sizes grow, improvements in associativity and tricks like memory compression are likely to have less of an effect: larger caches tend in general to be less susceptible to improvements of any other kind than further increases in size [Handy 1998].

Software management of cache replacement is likely to have stronger benefits as the cost of misses to DRAM increases. The biggest obstacle to managing caches in software is that the cost in lost instructions of a less effective replacement strategy has not, in the past, been high enough to justify the extra overhead of software miss handling. This obstacle is obviously reduced as the CPU-DRAM speed gap grows. Early work on software management of caches has addressed multiprocessor cache coherency, and there has since been work on software management of misses in virtually-addressed caches, to improve handling of page translation on a miss [Jacob and Mudge 1997].

The RAMpage model introduces the most comprehensive software management proposed yet, by managing the lowest-level SRAM as the main memory.

3.4 Summary

Caches are in the front line of the battle to bridge the CPU-DRAM speed gap. The strongest weapon is the least subtle – size. Nonetheless, given that caches have size limits for reasons like cost and packaging, addressing the memory wall implies that increasingly sophisticated approaches to improving caches are likely to become common.

Improving associativity – whether by hardware support, or a software approach like RAMpage – has obvious benefits. Less obvious is that increasing support for multithreading or multitasking can help to hide DRAM latency. However, there are sufficient studies of different approaches, including RAMpage, CMP and SMT which suggest that multithreading or other support for multiple tasks or processes will play an increasingly important role in latency hiding. Exploring these ideas further is the subject of the next section.

4 Multithreading and Multiple Process Support

4.1 Introduction

This section examines the relationship between hardware support for running multiple threads or processes and the memory wall. Multithreading refers to lightweight processes running in the same address space while multiple processes usually refers to processes which run in different address space, i.e., which have a higher overhead both in initial setup and in switching from one process to another.

Multithreading support on a uniprocessor has been introduced as a strategy for taking up the slack when a processor would otherwise stall [Tullsen et al. 1995], but multiple processors have not traditionally been seen in the same light. Accordingly, this section starts by examining more specifically how support for running multiple processes – whether lightweight or not – can address the memory wall. From this start, alternatives are examined – support for multiple threads or processes on one processor, versus multiple processors on a chip.

Finally, a summary concludes the section.

4.2 How it Addresses Memory Wall

At first sight, hardware support for running multiple threads is not an obvious candidate for addressing the memory wall. However, having other work available for the CPU when it is waiting for a cache miss means that the processor can be kept busy. Latency for the instruction causing the miss is not improved, but the overall effect can be improved responsiveness. However, the more obvious gain is in improved throughput, as the total work completed over a given time will be greater. The net effect is that overall performance is improved, and the extra latency is generally hidden.

It is even less obvious that a multiprocessor could address the memory wall. However, several processors with lower individual peak bandwidth than one aggressive design could see the same benefits as a multithreaded processor: there is more likely to be some useful work being done at any given time, since the probability that all processors would frequently experience a miss at the same time is likely to be low, even if the peak throughput is not being achieved.

4.3 Alternative Approaches

Two approaches are contrasted here: hardware support for multiple threads on the same processor, and chip multiprocessors. In addition, the RAMpage model, which does not require, but can use, hardware support for multiple threads or processes is used to illustrate the value of having alternative work available on a miss. RAMpage results are useful to make this point, as RAMpage has been measured with a multiprogramming workload, as opposed to the common practise in architecture research of measuring performance of applications singly.

Simultaneous multithreading (SMT) is the approach of supporting hardware contexts for more than one active thread. In the event of a cache miss (or any other pipeline stall, but cache misses are of most interest here), the processor can switch to another process without invoking the operating system. SMT systems can in principle have a variety of strategies for switching threads, from a priority system where there is a hierarchy of threads (the highest priority runs whenever possible, then the next, and so on) through to an implementation where all threads have the same priority. It is also possible to have variations on how fine-grained the multithreading is [Tullsen et al. 1995].

SMT has achieved some acceptance, though it was generally seen as a high-end server option, until the launch in 2002 of the multithreaded version of the Pentium 4.

Chip multiprocessors (CMP) are an alternative for achieving parallelism. Since an aggressive superscalar design cannot achieve its full throughput in general, using the same silicon to implement multiple simpler processors is an alternative [Olukotun et al. 1996].

CMP has the advantage that the design is relatively simple: instead of one, extremely complex processor, simpler design elements are replicated, with the potential of faster design turnaround, and lower probability of error. Further, as the total component count increases, scaling clock speed becomes progressively harder, unless the design is broken down into smaller, independent units.

Both SMT and CMP potentially require more memory bandwidth than a superscalar design taking the same chip space if claims of higher effective utilization of the CPU are correct. However, because more than one thread (or possibly process, in the case of CMP) is active, a cache miss can be tolerated because another thread remains active.

The RAMpage model is designed to allow for context switches on misses and owes its scalability in the face of the growing CPU-DRAM speed gap to its ability to do other work on a miss [Machanick 2000]. RAMpage results also demonstrate that one of the traditional trade-offs in cache design, reducing misses versus reducing average access time, can be eased by the ability to do alternative work on a miss. RAMpage with context switches on misses has relatively good results with large SRAM page sizes (equivalent to blocks in the lowest-level cache). These relatively large pages reduce misses, though not sufficiently to offset the time spent servicing the miss. However, the increased latency of misses is hidden by having other work available for the processor.

Figure 4.3 highlights how cache improvements are not enough.

The graphs show how simulated execution time for a workload consisting of a mix of 18 integer and floating-point benchmarks, totalling 1.1-billion references, is divided between levels of the memory hierarchy.

Looking from left to right in any one graph (except (e)–(f)), the figure illustrates how making higher-level caches larger (in this case, L1) may save execution time, but the net effect is that an increased fraction of run time is spent in DRAM. As the peak throughput of the processor increases versus DRAM's latency, the fraction of time spent waiting for DRAM increases. Looking down the figure, increases in L2 associativity as represented by introducing the RAMpage model without context switches on misses (Figures 2(c)–(d)) reduce the fraction of time spent waiting for DRAM, but the problem of the growing CPU-DRAM speed gap is not significantly addressed even with RAMpage's fully-associative SRAM main memory. Finally, Figures 2(e)–(f) illustrate how taking context switches on misses makes it possible to hide the latency of DRAM even across significant increases in the CPU-DRAM speed gap.

4.4 Summary

Based on experience with RAMpage, support for multiple threads or processes – whether on one CPU or by chip multiprocessors – has the potential to hide latency effectively, provided there is sufficient alternative work to fill the time while waiting for DRAM.

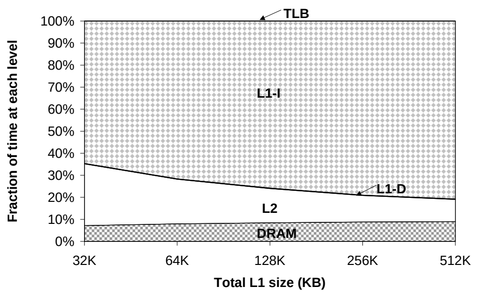
Results by researchers working on SMT designs confirm that, although the number of cache misses may go up, having alternative work available does help to hide the latency of DRAM. The same is likely to be true of CMP designs. The difference is that an SMT design can potentially keep one processor busy, whereas a CMP design may have one or more CPUs idle but could have better overall throughput than an aggressive superscalar design with a single thread of control.

The RAMpage model could combine with either SMT or CMP, and is hence not in competition with these alternatives.

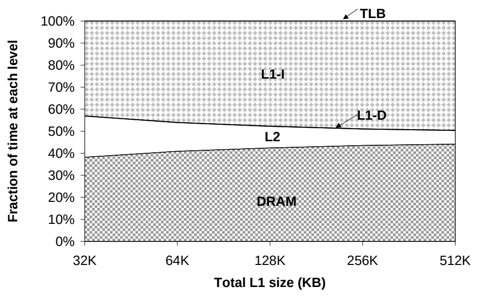
5 Conclusion

5.1 Introduction

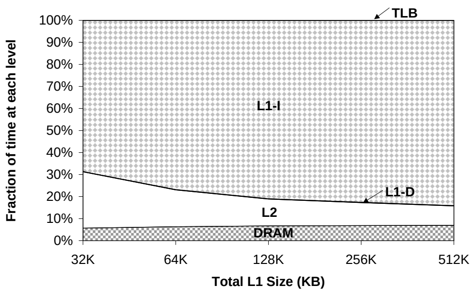
This paper has focused on how support for multiple threads or processes could contribute to solving the memory wall problem. The underlying assumption has been that there will be no great breakthrough in DRAM technology.



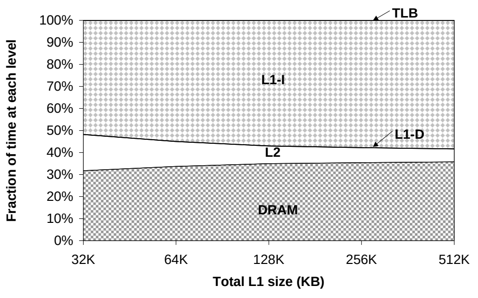
(a) Standard Hierarchy: 1 GHz



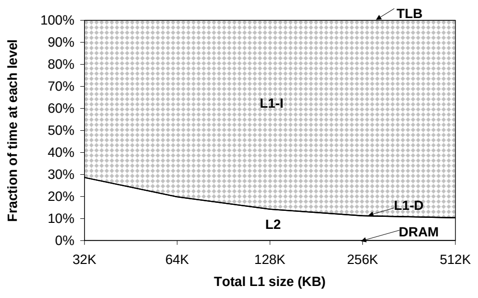
(b) Standard Hierarchy: 8 GHz



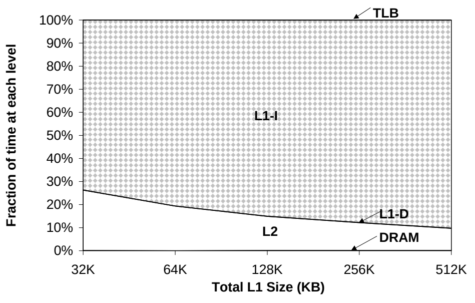
(c) RAMpage, no context switches on misses: 1 GHz



(d) RAMpage, no context switches on misses: 8 GHz



(e) RAMpage with context switches on misses: 1 GHz



(f) RAMpage with context switches on misses: 8 GHz

Figure 2: Fraction of time spent in each level of the hierarchy. *Figures in GHz are peak instruction throughput rate. TLB and L1 data (L1D) hits are fully pipelined, so they only account for a small fraction of total execution time; TLB misses are accounted for in other memory traffic. In all cases L2 (or RAMpage main memory) is 4 Mbytes.*

Of course, such assumptions are sometimes invalidated. For example, in the 1980s, the price improvement of disks was slower than the price improvement of DRAM, and it looked as if it was a matter of time before disks could be replaced by DRAM with a battery backup, or some other nonvolatile DRAM-based storage solution. However, disk manufacturers have improved their learning curve since 1990 to match that of DRAM (4 times the density every 3 years, as opposed to the old learning curve of doubling every 3 years), and consequently, magnetic disks remain the low-cost mass storage solution of choice on most computers.

The memory wall was already potentially a problem in 2002. Processors were available with a peak throughput of over 10-billion instructions per second (one every 0.1ns), while DRAM latency was still in the tens of nanoseconds, resulting in cache miss penalties in the hundreds of lost instructions.

The remainder of this section summarizes major issues and options identified in this paper, suggests a way ahead, and summarizes overall findings.

5.2 Major Issues and Options

A bigger issue not addressed in this paper is the need to look at an integrated solution, taking into account all aspects of design of a computer system which apply to memory performance, including multitasking, virtual memory, pipeline design, cache design, DRAM design and the hardware-software interface.

However, multitasking or multithreading alone has been shown to be a promising alternative: as the RAMpage model has shown, having additional work available while waiting for DRAM can make a significant difference. SMT studies have confirmed this effect, and there would appear to be a strong case for chip multiprocessors as an alternative to aggressively superscalar architectures.

Support for multiple threads or processes will likely be increasingly important, and mechanisms for switching to another thread or process on a miss to DRAM will be increasingly useful.

Given that most commonly-used operating systems (the various Microsoft offerings, and the various flavours of UNIX, including Linux and Mac OS X) have multithreading support, and have multithreaded user interfaces, there may be less work that it seems at first sight to exploit a capability of running another thread while waiting for DRAM. However, quantification of the value of this effect will require whole-system simulations, rather than the more common approach of simulating single applications.

5.3 Way Ahead

Work on parallelising compilers to find far parallelism and generate threads automatically out of sequential code should have increasing performance benefits as the need to find alternative work while waiting for DRAM increases. Work on finding threads has generally been focused on compilation, but there is no reason in principle that finding threads could not be done on existing compiled code. There is therefore potential for interesting future work on finding threads both at compile time, and in existing compiled code.

Increasing the degree of multithreadedness of code, in general, works towards the goal of finding alternative work for the processor while waiting for DRAM. Processor architects will then be in a position to focus more strongly on alternatives like SMT, CMP and RAMpage which can exploit multithreaded code to keep the processor busy while waiting for a miss to DRAM.

5.4 Overall Conclusion

Unless some big breakthrough in DRAM technology takes away the memory wall problem, improved approaches to addressing the problem become will increasingly important. If overall processor speed improves by a conservative 50% per year, while DRAM latency improves by 7% per year, the time to double the speed gap is just over 2 years, which represents a rapidly growing problem.

This paper has identified some promising directions for addressing the problem. Further, these directions in some cases may result in overall design simplification, so they do not represent impossible obstacles to dealing with the memory wall threat.

It will still be some time before we think of DRAM as a slow peripheral, but some change in mindset is likely to be needed soon, to break out of an unsustainable trend.

References

Chen, T. and Baer, J. (1992). Reducing memory latency via non-blocking and prefetching caches. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pages 51–61.

Crisp, R. (1997). Direct Rambus technology: The new main memory standard. *IEEE Micro*, 17(6):18–28.

Cuppu, V. and Jacob, B. (2001). Concurrency, latency, or system overhead: which has the largest impact on uniprocessor DRAM-system performance? In *Proc. 28th annual Int. Symp. on on Computer Architecture*, pages 62–71, Göteborg, Sweden.

Hallnor, E. G. and Reinhardt, S. K. (2000). A fully associative software-managed cache design. In *Proc. 27th Annual Int. Symp. on Computer Architecture*, pages 107–116, Vancouver, BC.

Handy, J. (1998). *The Cache Memory Book*. Academic Press, San Diego, CA, 2nd edition.

Jacob, B. and Mudge, T. (1997). Software-managed address translation. In *Proc. Third Int. Symp. on High-Performance Computer Architecture*, pages 156–167, San Antonio, TX.

Jouppi, N. (1990). Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Int. Symp. on Computer Architecture (ISCA '90)*, pages 364–373.

Jouppi, N. P. (1993). Cache write policies and performance. In *Proc. 20th annual Int. Symp. on Computer Architecture*, pages 191–201, San Diego, California, United States.

Kozyrakis, C., Perissakis, S., Patterson, D., Anderson, T., Asanović, K., Cardwell, N., Fromm, R., Golbus, J., Gribstad, B., Keeton, K., Thomas, R., Treuhaft, N., and Yelick, K. (1997). Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78.

Lee, J.-S., Hong, W.-K., and Kim, S.-D. (1999). Design and evaluation of a selective compressed memory system. In *Proc. IEEE Int. Conf. on Computer Design*, pages 184–191, Austin, TX.

Machanick, P. (2000). Scalability of the RAMpage memory hierarchy. *South African Computer Journal*, (25):68–73.

Mowry, T., Lam, M., and Gupta, A. (1992). Design and evaluation of a compiler algorithm for prefetching. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 62–73.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. (1996). The case for a single-chip multiprocessor. In *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 2–11, Cambridge, MA.

Rogers, A. and Li, K. (1992). Software support for speculative loads. In *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, Boston, Massachusetts, United States.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: maximizing on-chip parallelism. In *Proc. 22nd Annual Int. Symp. on Computer Architecture (ISCA '95)*, pages 392–403, S. Margherita Ligure, Italy.

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

Philip Machanick is a senior lecturer in the School of IT and Electrical Engineering at the University of Queensland, Australia. His research interests include computer memory systems, and Computer Science education. He holds a PhD from the University of Cape Town. He is a Senior Member of the IEEE, and a member of ACM. He can be reached at *philip@itee.uq.edu.au*.