Approaches to Addressing 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 the approaches separately, and investigates the extent to which approaches can work together. The major finding is that an integrated approach is necessary, considering all aspects of system design which can impact on memory performance. Specific areas which need to be addressed include models for allowing alternative work to be scheduled while the processor is waiting for DRAM, design of virtual memory management with minimizing DRAM references a major goal, and making it possible to apply increasingly sophisticated strategies to managing the contents of caches, to minimize misses to DRAM.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles — *cache memories/primary memory/virtual memory*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids; C.1.2 [**Processor Architectures**]: Multiprocessors; C.4 [**Performance of Systems**]: Design Studies
General Terms: processor, cache, main memory, virtual memory, translation lookaside buffer (TLB) Additional Key Words and Phrases: memory wall, CPU-DRAM speed gap

1 Introduction

The *memory wall* is defined as a situation where the much faster improvement of processor speed as compared with dynamic random access memory (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 [Moore 1965], leading to a doubling of CPU power every 18 months. Though some have raised the possibility of an end to this trend [Wilkes 1995], while others have started debating the consequences of an end to such progress [Cringely 2001], such an end is likely far enough off that planning for its continuation appears more reasonable than planning for its demise. Certainly, there are vigorous efforts being made to address predicted physical limits to progress [Hamilton 1999].

Since the mid 1980s, processor speed has generally improved by 50-100% per year, while DRAM speed has only improved at about 7% per year [Hennessy and Patterson 1996], resulting in a doubling of the speed gap between processor and main memory cycle time approximately every 6.2 years [Boland and Dollas 1994]. If the overall effect of speed improvements such as more aggressive pipelines in the CPU is taken into account, the speed gap doubles every 1 to 2 years.

The memory wall problem arises because the difference between two competing trends, each of which is an exponential function but with a different base, grows at an expontial rate. Specifically, if CPU speed improves by

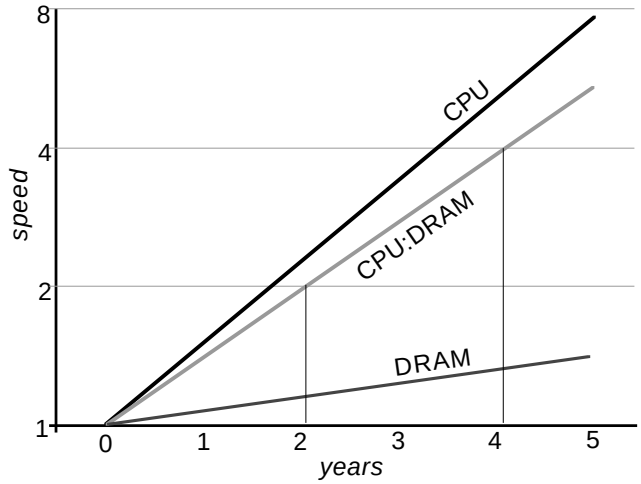


Figure 1: The growing CPU-DRAM speed gap, expressed as relative speed over time (log scale). *The vertical lines represent intervals at which the speed gap has doubled.*

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.

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.

To start with, some basic terminology is introduced, along with some terms relating to RAM, caches and processor architecture. Then, issues relating to each of DRAM, caches and processor architecture are outlined, as a basis for understanding the problem and proposed solutions, followed by an outline of the rest of the paper.

1.1 Definitions

Some terminology is introduced here; terms used in this paper are generally standard in the field, but definitions are supplied for the reader new to Computer Architecture. Terminology defined here is that which is useful for understanding the processor-memory interface. First, basic terminology common to a number of areas is introduced, followed by terminology specific, in turn, to processors, caches, DRAM (with some comparison to SRAM) and address translation.

Basic Terminology

The following are terms which are common across a number of areas:

- *latency* – time to complete a specific operation, also called *response time*; reducing latency is often an elusive goal. One of the biggest problems is that latencies accumulate, unless operations can be broken down into smaller units and performed in parallel.

- *bandwidth* – rate at which specified operations can be completed, also called *throughput*; improving bandwidth is usually easier than reducing latency, and the two goals can sometimes be traded for each other.
- *cycle time* – time for an operation to complete, typically as determined by a fixed, external clock; cycle time is the maximum time for any operation which has to be done within a clock cycle.
- *clock rate* – reciprocal of cycle time, measured in multiples of Hz.

Processors

Terminology in the area of processor design is reasonably standardised; definitions here are based on popular academic texts on computer organization [Patterson and Hennessy 1994] and architecture [Hennessy and Patterson 1996]:

- *central processing unit* – or *CPU*: synonymous with processor
- *arithmetic-logic unit* – or *ALU*: functional unit containing logic for arithmetic and control instructions; functional units for more specialised instructions such as floating point are separate parts of the CPU
- *register* – high-speed local storage tightly integrated into the processor
- *data path* – elements through which an instruction is processed, including the registers, ALU and memory unit
- *pipeline* – stages of the data path, each of which can execute in parallel, allowing more than one instruction to be active at a different *stage* of the pipeline simultaneously; if the pipeline is forced to wait for one or more clock cycles because of dependent instructions, a change in execution sequence or an unavailable resource, the pipeline is said to *stall*
- *instruction issue* – move instruction to the execute stage (or start of the execute stage, if it is split into multiple stages) of the pipeline
- *superscalar execution* – the ability to issue more than one instruction on the same clock cycle
- *instruction-level parallelism* – or *ILP*: ability to execute more than one instruction on the same clock cycle
- *branch prediction* – using history of previous behaviour to predict whether a branch is *taken* (change the sequence of execution as indicated) or not (continue to the next instruction)
- *speculative execution* – execution of a sequence of code which may turn out to be incorrect, usually through mispredicting a branch. Speculative execution implies the ability to undo misspeculation
- *out of order completion* – instruction complete before others which entered the pipeline before them
- *precise interrupt* – an interrupt which has the effect that all previous instructions execute to completion, and all following instructions have no effect

Caches

Cache terminology is well established [Smith 1982], with a few variations; the following are the most accepted variations:

- *line* – also called *block* – the minimum unit transferred to or from a cache
- *tag* – extra information associated with a cache line showing the state of the contents, including sufficient bits of the address to identify the contents: the tag may either be based on the physical or virtual address. Other information typically included is whether a block is *dirty*, i.e., modified from lower levels, *invalid*, meaning that there is nothing stored at that location. Requirements for additional state information vary (e.g., according to supported models for shared-memory multiprocessors)

- *indexing* – the operation of determining whether a given memory reference is found in the cache; caches may be physically indexed (requiring an address translation) or virtually indexed. The choice of indexing is orthogonal to how cache lines are addressed (e.g., it is possible to have a virtually indexed physically tagged cache [Kane and Heinrich 1992], though physically indexed and virtually tagged is an unlikely combination)

- *associativity* – a measure of the number of locations in which a given memory address may be placed. A *direct-mapped* cache has only one location for each line. A *fully-associative* cache can place any line anywhere, while an n*-way set-associative* cache is divided into *sets*, each containing n lines, and a given line can fall anywhere in a specific set

- *multilevel cache* – it is common to have one level of cache (*first-level cache* or *L1*) close to the CPU, operating as fast as possible, backed up by one or more levels of bigger, slower cache, labeled with higher numbers the further they are from the CPU; the *L2* cache is increasingly often on the same chip as the CPU

- *split cache* – first-level caches are often split into *instruction* and *data* caches, allowing different optimizations for the two kinds of references as well as simultaneous access (important for aggressive pipelines)

- *miss* – failure to find a reference in a cache: can be a *compulsory* or *cold start* miss resulting from no prior reference to the line containing the reference, *capacity* miss resulting from the cache being full or *conflict* miss resulting from the set the reference falls in being full

- *hit* – opposite of a miss: the required reference is in the cache

- *miss rate* – the fraction of references which miss; can be a *local* miss rate if there is more than one level of cache: the fraction of misses in relation to references seen at that level, or a *global* miss rate: the fraction of all references. Misses can also be split as *data* and *instruction* misses

- *replacement* – selection of an appropriate *victim* to eject when a miss occurs and all possible locations to place the block are occupied

DRAM versus SRAM

Dynamic random access memory (DRAM) is mainly distinguished from static random access memory (SRAM) in the underlying technology used to make a bit. DRAM uses a capacitor to store a bit which can be accessed by a single transistor, whereas SRAM uses a flip-flop, which is made of 4 to 6 transistors. Because the charge on a capacitor leaks, DRAM needs to be refreshed. Further, switching transistors is faster than changing the charge on a capacitor. Here is some standard terminology, again common in a variety of standard texts [Patterson and Hennessy 1994; Hennessy and Patterson 1996]:

- *refresh* – extra phase of the DRAM cycle required to maintain the charge on capacitors representing bits

- *row access* – DRAM is organised in a 2-dimensional structure; a row is first accessed using a *row access strobe* or *RAS* signal, then a specific bit by a *column access strobe* or *CAS*

- *page mode* – ability to access multiple columns after the row has been set up. Variations include *extended data out* or *EDO* and more recently *synchronous DRAM* or *SDRAM* which assumes that followup references will be to contiguous columns. SDRAM is faster than earlier page-mode DRAMs because, once the first column is addressed, subsequent references can be made every bus clock without futher addressing

- *Direct Rambus DRAM* – or *DRDRAM*: Rambus provides a packaging standard for both DRAM and its bus. Details include the abililty to transfer data on the leading and falling clock edge, a relatively high bus speed but a relatively narrow bus and – in the Direct Rambus generation [Crisp 1997] – the ability to pipeline multiple independent references.

- *bus clock* – in recent designs, the bus clock speed is a significant indicator of the speed of DRAM. SDRAM timing includes the number of bus clocks to start initiating a reference and the time to do each subsequent reference. Earlier designs did one transfer per clock; Rambus and *double datarate DRAM* or *DDRDRAM* transfer on both the rising and falling clock edges.

- *SRAM timing* – SRAM cycle time is generally given as the time for the data to be ready after initiation of the read or write operation. As with DRAM, burst modes are available, and combinations such as synchronous and double data rate have existed at roughly the same times and with the same benefits as the equivalent DRAM variations. The speed gain in the case of SRAM is not as great, but the same principle applies: SRAM is fastest when doing sequential accesses to increasing addresses

Address Translation

This terminology too is standard across a variety of architecture and operating systems books [Patterson and Hennessy 1994; Hennessy and Patterson 1996; Maekawa et al. 1987; Crowley 1997; Silberschatz et al. 2002]:

- *virtual address* – the address as contained in a program (with the exception of some operating system code, which is physically addressed) which allows a program to be partially and noncontiguously loaded into memory, usually in fixed-size units called *pages*
- *page table* – table of address translations from virtual to physical. Since a page table may be very large, it can be itself in the virtual address space and hence susceptible to being paged out to disk
- *inverted page table* – page table with one entry per physical address, accessed by a hash function on the virtual address
- *translation lookaside buffer* – or *TLB*: (usually) small cache of recent page translations. As with other caches, the TLB can and often is split between instruction and data entries. Misses from the TLB can be handled with hardware support in the case where the missing entry can be found relatively easily but in the worst case software intervention is necessary (e.g., the page table entry required may have been paged to disk)

1.2 DRAM Issues

The biggest issue in understanding the role of DRAM in the memory wall problem is the relative rate of improvement of DRAM as opposed to other elements of the system hierarchy. Historically, the big driving factor in DRAM improvement has been capacity, rather than speed. DRAM, since the 1970s, has improved in capacity at a rate of a factor of 3 every 4 years [Hennessy and Patterson 1996]. Consequently, the role of virtual memory has changed somewhat since the 1960s, when its major purpose was to provide a significantly larger address space than was affordable even in large-scale computers. The first successful virtual memory design, the Atlas, was initially configured with 6 times as much backing store as main memory [Kilburn et al. 1962]. While more recent designs may in some cases rely on a significantly larger virtual than physical address space, the significant growth in the speed gap between backing store and main memory means that a high page fault rate cannot be tolerated.

In 2002, a fast disk had an access time of 5ms, while a fast DRAM could be accessed in tens of nanoseconds, a speed difference of a factor of around 10^6. This should be contrasted with the relative speeds in the early 1960s [Kilburn et al. 1962] of a drum secondary memory (about the same as a fast disk 40 years later) and DRAM of the time (about 1μs), resulting in a speed ratio of about 5000. Typical instructions on the Atlas took about 2μs to complete, so a page fault cost roughly 2500 instructions.

Going back to 2002 numbers, a commodity processor running at 2GHz capable of completing 8 instructions per cycle would complete an instruction every 0.0625ns, roughly 300 times faster than the fastest cycle time of available DRAM at the time, and about 500 times the effective DRAM cycle time of common DRAM in early 2002. Given that a cache miss typically involves several DRAM accesses, there is potential for a cache miss to DRAM to approach the times (in lost instructions) of early virtual memory machines not long after 2002.

Consequently, the memory wall problem has remained an issue since it was first raised in 1997, and approaches to addressing the problem have continued to be of significance.

1.3 Cache Issues

Caches are generally made of static RAM (SRAM), and, increasingly, at least 2 levels of cache are being integrated onto the processor chip. It becomes increasingly useful to split the on-chip cache into two or more levels, as the amount which fits on the chip is grows. It is harder to make a larger cache fast and optimizing for the best hit time does not always result in the lowest possible miss rate [Handy 1998]. Consequently, the first-level (L1) cache, which needs to keep up with the processor, maybe be designed with trade-offs which make speed easier to achieve

(smaller size, lower associativity, split instruction and data cache) and be backed up by an L2 cache which is optimized for lower misses (larger size, higher associativity, unified instruction and data cache).

Optimizations which can apply at the first level include various strategies for increasing the support for multiple simultaneous access of the cache, from splitting the instruction and data caches, through to various variations on multiporting or having multiple banks in the cache [Rivers et al. 1997].

Generally speaking, all variations on a design which allow choice in respect of where a reference is found, or between multiple references occurring simultaneously, are going to add extra stages of logic to a cache reference and therefore potentially slow the cache down. A smaller cache can use faster components and have shorter overall signal propagation, and hence be faster, but this kind of design for speed does not scale up [Handy 1998]. The maximum propagation delay increases with size, and heat dissipation becomes an increasing problem with high density packaging of fast components.

Placing the L2 cache on the processor chip, aside from relieving pressure to increase the size of the L1 cache, in the face of requirements to make it as fast as possible, has several packaging benefits. First, off-chip delays are avoided. Second, the need for having a very fast bus outside the chip is avoided. Third, design tolerances on-chip are much tighter than off-chip, so the need to design for variations, e.g., in bus timing are reduced. Finally, a specialized, high-speed SRAM design can be used, since the cache is not a separate component which systems designers would like to buy from a standard parts catalogue.

Caches play an important role in reducing the effect of the memory wall. SRAM speed improvement can generally be expected to track CPU clock speed, since it is essentially based on the same technology. Processor speed improvement generally can be expected to be more than clock speed improvement because of increases in ILP and other architectural improvements, but CPU designers have to find approaches such as those outlined here to make it possible for the L1 cache to keep up, otherwise their proposed speed improvement will not be achieved. L2 caches on the other hand may run at a fraction of CPU clock speed, depending on how aggressive the design.

Caches are able to take advantage of some advances in DRAM, specifically, the availability of higher-speed burst-mode transfers. Since a cache block is usually several memory bus widths (cache blocks typically range from 32 to 128 bytes; memory buses range from 16 to 128 bits, or 2 to 16 bytes), such burst mode transfers (as for example in SDRAM or Rambus) are a good fit to requirements of a cache. However, the value in burst modes lies in moving a relatively large amount of data at a time to amortize the startup cost; exploiting this property of a DRAM by having relatively large cache blocks further adds to the latency of handling a miss.

Caches, therefore, are able – with some limitations – to scale in speed with CPU speed improvement. To a lesser extend, caches can scale up in size to mask the CPU-DRAM speed gap, but packaging constraints limit how large a cache can be built which is anywhere close to the speed of a fast 2002 or later processor.

Consequently, examining improvements to caches is an important aspect of understanding approaches to addressing the memory wall, but cannot be a complete solution.

1.4 Processor Architecture Issues

If design of memory components presents problems, a major source of the problems is the memory-referencing behaviour of the CPU.

A processor executing a single, sequential instruction stream is difficult enough, but instruction-level parallelism adds to the difficulties.

DRAM, as has been outlined, works best in burst mode, when memory accesses are contiguous. Caches can be designed around this requirement with blocks (lines) which require several memory references to fill them. The processor, on the other hand, does not necessarily execute instructions in sequence, or refer to data at sequentially-increasing addresses.

In many programs, the size of a basic block (a sequence of code with one entry point and one exit point) is on average less than 6 [Wall 1991], so finding instruction-level parallelism within strictly sequential code is not likely to lead to good results, especially as instructions within a basic bock may have dependences between them. Consequently, aggressively superscalar architectures employ a variety of techniques to attempt to find parallelism outside known paths of control, such as branch prediction [Yeh and Patt 1993; Young and Smith 1999] which, combined with speculative execution [Hiraki et al. 1998], allows the processor to run ahead of the point where it is known which alternative branch outcome will occur.

Unfortunately, branches are bad for memory performance, as they do not exploit available burst modes. Speculative execution, which may involve more than one wrong path before branches are resolved, add to the complexity of matching the CPU to memory characteristics.

Consequently, caches, while offering a respite from dealing directly with the CPU-DRAM speed gap, are also faced with design pressures on the CPU side. To address the memory wall fully, therefore, requires that attention be paid to cache design not only from the perspective of bridging the speed gap to DRAM, but also from the perspective of dealing with the mismatch between memory characteristics and the requirements of an aggressively superscalar CPU.

1.5 Structure of Paper

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 is relevant because it offers an option for keeping the processor busy while waiting for DRAM.

Section 5 presents issues in address translation, and explains why they are relevant to the memory wall problems. Some solutions are presented.

Instruction-level parallelism (ILP) is evaluated in Section 6 in relation to the memory wall. Achieving high ILP is a design goal of almost all recent microprocessor designs, so it is useful to evaluate whether high ILP is a useful goal as DRAM accesses become a higher overhead.

Section 7 evaluates the alternatives, and isolates the most promising and least promising alternatives, as a basis for future research.

Finally, Section 8 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, techniques for hiding the latency of DRAM, 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 [Lin et al. 2001; 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 [Fromm et al. 1997] 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 [Hidaka et al. 1990]; this idea has been taken up more recently in the DDR2 standard [Davis et al. 2000].

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 1.1 to 2 years.

2.3 Hiding Latency

Hiding latency is not a new idea: shared-memory multiprocessors have exposed problems of DRAM latency long before the memory wall problem was raised. Shared-memory multiprocessors, in addition to the usual problems of the CPU-DRAM speed gap, have overheads of sharing. These overheads include a longer memory bus, protocols to permit sharing while keeping memory consistent and additional bus traffic caused by sharing.

Some of the ideas to hide latency in the shared-memory world [Dahlgren et al. 1994; Lenoski et al. 1992] should be expected to find their way into the uniprocessor world, as indeed some have.

Some of these ideas include:

- *prefetch* – loading a cache block before it is requested, either by hardware [Chen 1995; Kroft 1981] or with compiler support [Mowry et al. 1992]; predictive prefetch attempts to improve accuracy of prefetch for relatively varied memory access patterns [Alexander and Kedem 1996]
- *nonblocking cache* – a cache which can continue servicing other requests while waiting for a miss [Kroft 1981; Chen and Baer 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

These ideas have downsides. For example, any form of prefetch carries the risk of replacing required content from a cache sooner than necessary, creating the opposite effect to that intended. A nonblocking cache obviously requires an increase in hardware complexity, 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.

2.4 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 sequential 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 on-chip caches, and pipelining multiple references. However, completely random references must ultimately result in the processor seeing the full latency of a DRAM reference.

2.5 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. Some issues which could be covered under this heading have been addressed as part of the discussion of hiding DRAM latency in Section 2.3.

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

Caches can improve tolerance of increasing latency by increasingly aggressive prefetch strategies, as outlined in Section 2.3. Another approach to tolerating latency is to have other work for the processor on a stall; *simultanous multithreading* or *SMT* is such an approach [Lo et al. 1997]; SMT is examined in more detail in Section 4.

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

- *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]

All of these 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 [IBM 1998]; 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 [AMD 2001]. 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 [Machanick et al. 1998]. 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.

There have been attempts at implementing higher associativity with a best-case hit time similar to that of a direct-mapped cache, with an extra penalty for other hits. An example of this strategy is the column-associative cache [Agarwal and Pudar 1993], which uses hash functions to find alternative locations for blocks as an extension of the design of a direct-mapped cache. In the best case, a hit is the same as in a direct-mapped cache, with a penalty for finding a block in an alternative location. A column-associative cache is a simpler design than an indirect index cache. While a column-associative cache achieves approximately the same miss behaviour as a 2-way associative cache, rather than a fully-associative cache, it likely has a lower average hit time than an IIC. However, an IIC is designed for software-managed replacement, in addition to supporting full associativity [Hallnor and Reinhardt 2000], so it should generally have a lower miss rate than a column-associative cache.

There are various other schemes for reducing conflict misses, such as indexing caches using hash functions [Topham et al. 1997] and hardware to support advice to the operating system to place pages in such a way as to minimize conflicts [Bershad et al. 1994]. Such approaches, while not necessarily directly increasing associativity, have the same effect, although they cannot decrease conflict misses as much as full associativity would.

Memory compression can also reduce misses, by increasing the effective capacity of the cache [Lee et al. 1999].

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 was designed to support multiprocessor cache coherency [Cheriton et al. 1986], 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 [Silberschatz et al. 2002].

Multithreading support on a uniprocessor has been introduced as a strategy for taking up the slack when a processor would otherwise stall for a cache miss [Lo et al. 1997], 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 [Lo et al. 1997].

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 of all processors experience a miss at once 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.

Simultaneous multithreading (SMT) is the approach of supporting hardware contexts for more than one active process. 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, through studies which have shown it to be a win with a variety of workloads such as database [Lo et al. 1998]. There have been several studies on detail of the implementation including register allocation strategies [Waldspurger and Weihl 1993; Hidaka et al. 1993].

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 [Ho et al. 2001].

One of the difficulties with CMP is finding threads in nonthreaded code. One approach to the problem is to use a compiler which generates *speculative threads*: threads which may be incorrect to run in parallel because of dependences (e.g., caused by pointers) which can only be detected at run time [Krishnan and Torrellas 1999].

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 by switching to another thread.

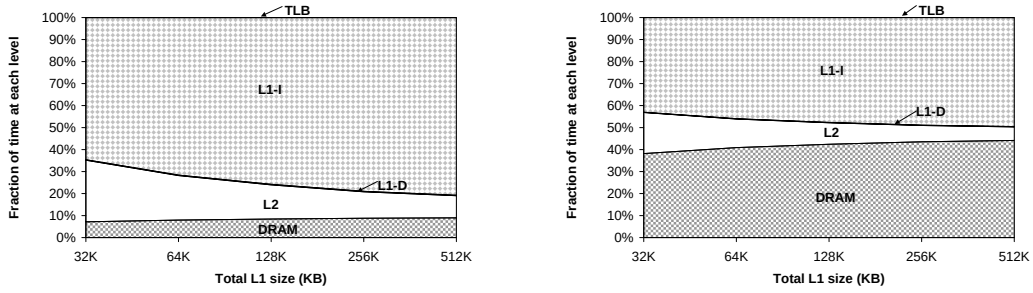
The RAMpage model is designed to allow for context switches on misses [Machanick et al. 1998] and indeed owes its scalability in the face of the growing CPU-DRAM speed gap to its ability to do other work on a miss [Machanick 2000]. RAMpage results also demonstrate that one of the traditional trade-offs 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 [Machanick et al. 1998].

Figure 4.3 highlights how cache improvements are not enough. Looking from left to right in any one graph (except (e)–(f)), the figure illustrates how making 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) illustrates 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

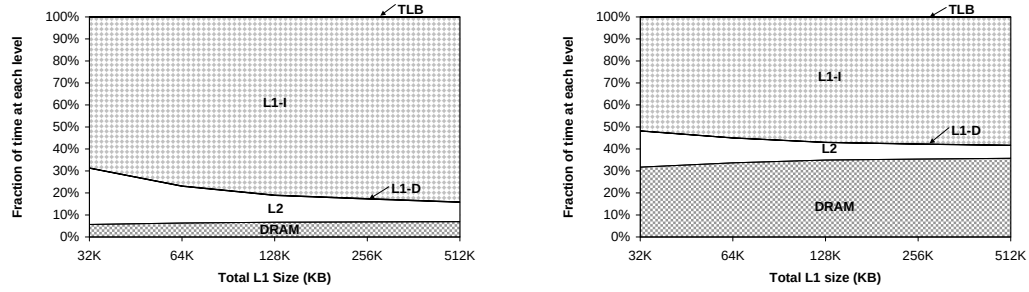
Based on experience with RAMpage, support for multiple threads or processes – whether on one CPU or by chip multiprocessors – has the potential to hide latency effectively, provided there is sufficient alternative work to fill the time while waiting for DRAM.

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.



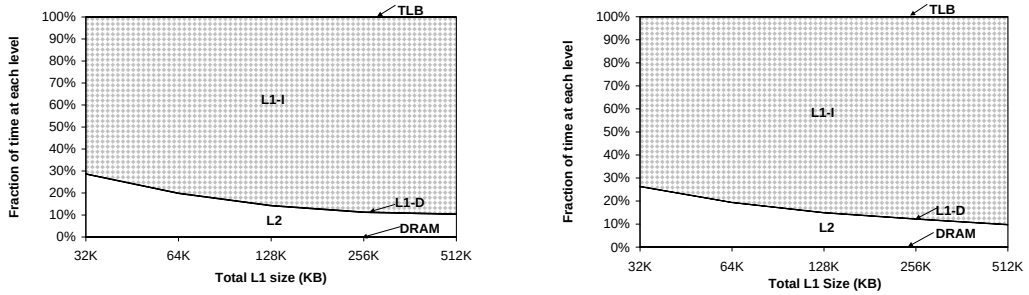
(a) Standard Hierarchy: 1 GHz

(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.*

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

Further research on memory effects of multithreading and multitasking would be useful to make the contribution of these ideas to addressing the memory wall clearer.

5 Role of Address Translation

5.1 Introduction

Address translation is often paid insufficient attention in design of memory hierarchies for performance, because it is easy to overlook as an orthogonal concern. Since every memory reference (instruction fetch as well as data reads and writes) in ordinary programs needs to be translated, it is essential that address translation be as quick as possible, hence the use of a high-speed cache of recent page translations, the translation lookaside buffer (TLB).

Organization of the page table, and mechanisms for finding a missing TLB entry in a page table, are key aspects of performance of a virtual memory system.

Clearly, minimizing page faults is also a major issue. However, in terms of the memory wall, this issue is not a factor, since the concern here is in minimising references to DRAM. Avoiding page faults to disk is of course important, but beyond the scope of this paper. Accordingly, the focus in this section is on page table organization and TLB management.

5.2 Page Tables

Page table organization can be divided into two major categories, with further subdivision within those cageories: forward-mapped and inverted page tables.

A forward-mapped page table is organized by virtual address. Logically, it is an array indexed by virtual address (and therefore, there is one page table per virtual address space). Each entry contains a physical translation for the virtual page, plus any tag bits or state information required by the operating system (e.g., whether the page is dirty, whether it is present in physical memory, and its location on disk).

Forward-mapped tables are often organized hierarchically, so that the whole page table need not be in physical memory at once. Depending on the organization, some designs make it possible in the best case to find a translation on one probe; others may require at least 2 data references. Curiously, Intel's page table design which requires at least 2 data references generally outperformed other designs in one study [Jacob and Mudge 1998]. However, the fact that the TLB was managed in hardware may have been a contributing factor.

An inverted page table has one entry per physical page; virtual to physical translations are looked up by hashing on the virtual address. An inverted page table works well with a relatively sparse virtual address space, or when the virtual address space is many times the size of the physical address space [Huck and Hays 1993]. Although an inverted page table needs more information per entry (a hash table needs to handle collisions), in practice, the amount of memory needed at higher levels of the memory hierarchy is lower than for a forward-mapped table, because unneeded page table entries are not used [Jacob and Mudge 1998].

5.3 Translation Lookaside Buffer

Some have investigated TLB behaviour and possible improvements [Kandiraju and Sivasubramaniam 2002], while others have noted the performance costs of poor TLB performance [Huck and Hays 1993], and have suggested workarounds [Cheriton et al. 1993]. The major source of poor performance is TLB misses. A TLB miss is on the critical path of the CPU, in that it prevents either an instruction fetch or a data reference from completing. Depending on where the page table entry is found and how the page table is organized, handling a TLB miss can take a minimum of 1 or 2 data references [Jacob and Mudge 1998]. However, over and above the data references, unless the TLB is managed in hardware, there will also be instruction references, and the overheads of interrupting the pipeline and passing control to the operating system. If the page table entry is found in the L1 cache, the expense of a hardware-handled miss is not high but page tables, in general, can be anywhere in the memory hierarchy. In the worst case, a TLB miss can result in a page fault and, on average, may cost as much as, or more than a cache miss to DRAM.

Clearly, if TLB misses are so expensive, effort to reduce or remove TLB misses is worthwhile.

5.4 Performance Issues

The biggest performance issues in page translation, as far as the memory wall is concerned, arise out of TLB behaviour, since handling TLB misses can result in references to DRAM. In some studies, handling TLB misses has accounted for as much as 40% of run time [Huck and Hays 1993], with figures in the region of 20–30% not uncommon [Cheriton et al. 1993; Machanick 1996].

Further, virtual memory overheads tend to be underestimated, because the effect of extra cache misses generated by virtual memory management are difficult to isolate. According to one study, such overheads approximately double the fraction of time spent managing page translation over earlier estimates [Jacob and Mudge 1998].

A comprehensive solution to the memory wall problem, therefore, must take into account virtual memory overheads.

5.5 Solutions

There are several approaches to reducing virtual memory overheads. First, if TLB management is a problem, the obvious solution is to design a system which does not need a TLB. Second, if extra cache misses generated by virtual memory management are a problem, memory references used to manage page tables and the TLB need to be minimized. Finally, page tables can be organized so as to minimize the need to go to slower levels of the hierarchy.

A TLB is needed because address translation is required to index a cache, and to check its tag. A virtually-addressed cache [Inouye et al. 1992] eliminates the need to do address translation, except on a cache miss. The TLB can either be eliminated completely, or only used on a cache miss. Unfortunately, a virtually-addressed cache introduces a number of problems, such as handling aliases and context switches. While these problems can be solved [Wheeler and Bershad 1992], virtually addressed caches have remained the exception.

Minimizing cache misses when handling page table management can be achieved in part by hardware management of a page table, and also by use of an inverted page table [Jacob and Mudge 1998]. Hardware management eliminates instruction references (though data references still exist) and, possibly more importantly, the need to invoke the operating system. An inverted page table, while needing larger entries, does not pollute the cache with unused page-table entries.

The RAMpage hierarchy uses an inverted page table in its SRAM main memory, to avoid having to take a TLB miss to DRAM (provided the reference resulting in the miss is itself found in one of the SRAM levels). The RAMpage hierarchy is therefore somewhat more tolerant of TLB misses than a conventional hierachy, as evidenced by its increased scalability as the CPU-DRAM speed gap grows [Machanick 2000]. The RAMpage hierarchy would be further improved with a hardware page table walker.

5.6 Summary

Virtual memory overheads are an important, if sometimes neglected, component of the memory wall problem. While traditional page translation has been seen as a problem of minimizing referenecs to disk, TLB misses and other causes of memory references while managing page tables must be taken into account as part of any strategy to minimize loss of performance as a result of increases in DRAM latency (as measured relative to processor performance).

6 Instruction-Level Parallelism

6.1 Introduction

Aggressive pursuit of instruction-level parallelism is a common design goal. In the commodity processor market, in 1999, it became possible to buy a CPU which was 9-way superscalar [AMD 2001]. Further, Intel's IA-64 design (in its first commercial incarnation called the Itanium) was designed to develop the pursuit of instruction-level parallelism further, with design features to enable the compiler to provide hints to the hardware as to where parallelism was to be found [Dulong 1998].

Given the threat of the memory wall, it is worth considering whether the aggressive pursuit of instruction-level parallelism is sustainable. This section briefly reviews a few design problems which instruction-level parallelism poses for the memory system, and relates those problems to the growing CPU-DRAM speed gap.

6.2 Design Problems

Aggressive ILP causes many design problems arising out of the fact that code is often inherently sequential at a fine-grained level, as a result of the inherently sequential nature of the typical instruction set architecture in common use, as well as the inherently sequential programming languages in common use.

Here, however, only a few design problems which directly relate to memory use are addressed, since ILP is a large field, and much of it is beyond the scope of this paper.

First, aggressive instruction-level parallelism generally requires that a significantly larger window of code be fetched than can be issued on one clock cycle, since there may be dependences which prevent all contiguous instructions in a given group from executing simultaneously. The more aggressive the intended ILP, the bigger the window is likely to need to be over and above the actual number of needed instructions [Nakatani and Ebcioğlu 1990]. Potentially, a significant fraction of instructions fetched in such a large window will not be needed because of branches and other changes in sequence. Those extra instructions may displace other content from the cache which may be needed soon, causing unnecessary misses; as the window becomes larger, increasingly complex branch prediction becomes necessary to offset these negative effects [Skadron et al. 1999]. In any case, some studies have found that there is significantly more parallelism available too far apart for a hardware window containing such *far parallelism* to be practical [Martel et al. 1999; Postiff et al. 1999].

Second, aggressive ILP increasingly relies on speculative execution, since branch outcomes may depend on instructions close to the branch. Such speculative execution, in the case where a wrong path is taken, again may load unnecessary instructions in the cache, causing unnecessary misses. Such unnecessary misses need to be weighed against the prefetch effect of the instruction fetches which may not be needed in the short-term, but which may be useful later. Increasingly complex strategies become necessary, in the face of higher effective memory latencies, for more aggressive pipelines to have a net win, once memory effects are taken into account [Lee et al. 1995].

6.3 Relation to Memory Wall

Increasingly aggressive ILP is not only problematic from the point of view of making pipeline design increasingly difficult, including the need for complex mechanisms like speculative execution, but it stresses the memory system by requiring potentially unneeded memory references (large instruction windows, misspeculated wrong paths). While some of these unneeded references may have a useful prefetch effect, ensuring that the overall effect is useful becomes increasingly difficult as the CPU-DRAM speed gap grows. While the relatively low latency of interfaces to on-chip caches is some help, any unneeded reference which causes an extra miss to DRAM is increasing cause for concern.

Approaches to improving parallelism which allow alternative work to be done on a cache miss, and which do not aggressively load potentially unwanted data and instructions into caches, are more likely candidates for improving performance in the face of the memory wall.

Clock speeds and peak instruction issue rates in 2002 have already exceeded rates at which RAMpage studies estimate that the RAMpage model with context switches on misses could produce an improvement of over 60% in performance [Machanick 2000]. Even more aggressively superscalar approaches, therefore, have to be contrasted with other forms of parallelism which can permit other work to be done on a miss to DRAM.

6.4 Summary

Aggressive pursuit of instruction-level parallelism is in general unsustainable, if research results relating to available local parallelism are not overturned. Since such aggressive ILP stresses the memory system, other strategies for improving performance seem worth considering.

While SMT may aid aggressive ILP-based designs to achieve a reasonable throughput, CMP is an alternative design strategy for achieving parallelism which may win favour, given the fact that far parallelism is not adequately exploited by a focus on ILP. CMP, however, would make a stronger case if it were paired with a model for finding alternative work on a miss to DRAM.

7 Evaluation of Alternatives

7.1 Introduction

On the assumption that the memory wall problem is not going to go away, this section summarizes approaches which are likely to be effective, and those which are less likely to be effective, based on previous discussion.

In general, approaches which are more comprehensive seem more useful to pursue. Further, approaches which avoid hardware complexity seem more likely to succeed, if only because they are more likely to win any race to bring out a new implementation.

7.2 Effective Approaches

Improvements to caches, including increasing associativity (or effective associativity) appear to be worth pursuing, at least for the short term. As available chip space grows, it is worth putting larger (but possibly slower) caches on chip. Any loss in speed through (for example) higher associativity can to some extent be offset by the gain of avoiding off-chip latencies.

Of the approaches to cache improvement considered in this paper, indirect index caches (IIC) and RAMpage appear to be the most promising. IIC solves the problem of full associativity with relatively modest hardware, and avoids the software overheads of RAMpage page replacement in its management of the SRAM-DRAM interface. RAMpage also has the drawback of requiring operating system changes to the virtual memory system. However, RAMpage has some advantages, including managing the TLB purely in SRAM layers of the memory system, and the potential to take context switches on misses. RAMpage also does not require complex hardware: it uses a simple SRAM addressing scheme, and can use a standard TLB.

Following from RAMpage results, it would appear that any support for multithreading or multiple processes has the potential to take up the slack while the processor is not busy. Simultaneous multithreading (SMT) and chip multiprocessors (CMP) are both candidates. For CMP to be effective, automated generation of multiple threads needs to become more commonplace. However, many existing workloads are already multithreaded (for example, all popular operating systems have reasonable support for multithreaded applications, and have multithreaded user interfaces).

7.3 Less Effective Approaches

In general, aggressive pursuit of ILP appears to be in conflict with addressing the memory wall problem. Adding SMT can potentially alleviate the problem, by ensuring that the processor always has some work, but that is layering complexity on top of complexity.

The IRAM approach is likely to have some benefits as once-off fix, but once off-chip latency is removed, the fix cannot be applied again. Further, limiting the amount of RAM per processor limits design flexibility. It seems likely that, other than in embedded applications where resources can be fixed in advance, an IRAM machine will have to have the capability of adding off-chip RAM, which will reintroduce off-chip latency.

7.4 Summary

The most important single strategy for addressing the memory wall is likely to be having some model for the processor to do other work while waiting for DRAM. All other approaches only stave off the point where DRAM becomes a performance bottleneck.

8 Conclusion

8.1 Introduction

This paper has focused on issues which are most likely to provide solutions to the memory wall problem. The underlying assumption has been that there will be no great breakthrough in DRAM technology. 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 [Hennessy and Patterson 1996]), 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.

It is therefore important that serious alternatives be evaluated. Much computer architecture research is done in isolation: the pipeline is evaluated, the first-level cache is evaluated, or some other area of memory hierarchy is considered on its own. To make significant progress on the memory wall problem, an overall solution needs to be considered which looks at the overall system, not just specific components in isolation. This paper has attempted to pull together work from a variety of areas, to arrive at a comprehensive understanding of the problem and potential solutions.

The remainder of this section summarizes major issues identified in this paper, re-examines promising alternatives, and suggests a way ahead.

8.2 Major Issues

The biggest issue identified 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.

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.

Virtual memory management is a significant factor in performance and, as DRAM latency (in effect) grows, managing all aspects of the VM subsystem, but most particularly the TLB, will likely assume increasing importance.

Ensuring that all these components of system design interact in the best way for minimizing DRAM latency implies that performance studies only investigating a single aspect of system performance need to be considered with some care.

8.3 Promising Alternatives

The most promising alternatives identified here require considering a range of design improvements.

The virtual memory system should be designed to minimize references to lower levels of the hierarchy, as well as cache pollution by VM manangement. That combination suggests an inverted page table with hardware support for TLB miss handling.

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.

Finally, some sort of software management of the cache-DRAM interface is likely to become increasingly desirable, as the cost of poor decisions on replacement to DRAM increases.

8.4 Way Ahead

Work on parallelising compilers to find far parallelism and generate threads automatically out of sequential code [Martel et al. 1999] should have increasing performance as the need to have alternative work for the processor 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.

8.5 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. Although the CPU-DRAM speed gap has previously been estimated to double approximately every 6.2 years, this was based on clock speed – a conservative estimate, given that implemenation improvements (deeper and wider pipelines, speculative execution, etc.) have also boosted performance. If overall processor speed improves by 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

Agarwal, A. and Pudar, S. (1993). Column-associative caches: A technique for reducing the miss rate of direct mapped caches. In *Proc. 20th Int. Symp. on Computer Architecture (ISCA '93)*, pages 179–190.

Alexander, T. and Kedem, G. (1996). Distributed prefetch-buffer/cache design for high-performance memory systems. In *Proc. 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA)*, pages 254–263, San Jose, CA.

AMD. *AMD Athlon Processor Model 4 Data Sheet* [online]. (2001). Available from World Wide Web: `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23792.pdf`.

Bershad, B., Lee, D., Romer, T., and Chen, J. (1994). Avoiding conflict misses dynamically in large direct-mapped caches. In *Proc. 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-6)*, pages 158–170.

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

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.

Chen, T.-F. (1995). An effective programmable prefetch engine for on-chip caches. In *Proc. 28th Int. Symp. on Microarchitecture (MICRO-28)*, pages 237–242, Ann Arbor, MI.

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

Cheriton, D., Slavenburg, G., and Boyle, P. (1986). Software-controlled caches in the VMP multiprocessor. In *Proc. 13th Int. Symp. on Computer Architecture (ISCA '86)*, pages 366–374, Tokyo.

Cringely, R. X. (2001). Be absolute for death: life after Moore's law. *Communications of the ACM*, 44(3):94.

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

Crowley, C. (1997). *Operating Systems: A Design-Oriented Approach*. Irwin, Chicago.

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.

Dahlgren, F., Dubois, M., and Stenström, P. (1994). Combined performance gains of simple cache protocol extensions. In *Proc. 21st annual Int. Symp. on Computer Architecture*, pages 187–197, Chicago, Ill.

Davis, B., Mudge, T., Jacob, B., and Cuppu, V. (2000). DDR2 and low latency variants. In *Solving the Memory Wall Problem Workshop*, Vancouver, Canada. In conjunction with 26th Annual lnt. Symp. on Computer Architecture.

Dulong, C. (1998). The IA-64 architecture at work. *Computer*, 31(7):24–32.

Fromm, R., Perissakis, S., Cardwell, N., Kozyrakis, C., McGaughy, B., Patterson, D., Anderson, T., and Yelick, K. (1997). The energy efficiency of IRAM architectures. In *Proc. 24th Int. Symp. on Computer Architecture*, pages 327–337, Denver, CO.

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.

Hamilton, S. (1999). Semiconductor research corporation: Taking Moore's Law into the next century. *Computer*, 32(1):43–48.

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

Hennessy, J. and Patterson, D. (1996). *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann, San Francisco, CA, 2nd edition.

Hidaka, H., Matsuda, Y., Asakura, M., and Fujishima, K. (1990). The Cache DRAM architecture: A DRAM with an on-chip cache memory. *IEEE Micro*, 10(2):14–25.

Hidaka, Y., Koike, H., and Tanaka, H. (1993). Multiple threads in cyclic register windows. In *Proc. 20th Annual Int. Symp. on Computer Architecture (ISCA '93)*, pages 131–142, San Diego, CA.

Hiraki, K., Tamatsukuri, J., and Matsumoto, T. (1998). Speculative execution model with duplication. In *Proc. 1998 Int. Conf. on Supercomputing*, pages 321–328, Melbourne, Australia.

Ho, R., Mai, K. W., and Horowitz, M. A. (2001). The future of wires. *Proc. of the IEEE*, 89(4):14–25.

Huck, J. and Hays, J. (1993). Architectural support for translation table management in large address space machines. In *Proc. 20th Int. Symp. on Computer Architecture (ISCA '93)*, pages 39–50, San Diego, CA.

IBM. *PowerPC 750 RISC Microprocessor Technical Summary* [online]. (1998). Available from World Wide Web: `http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699300470399/$file/750_ts.pdf`.

Inouye, J., Konuru, R., Walpole, J., and Sears, B. (1992). *The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance*. Tech. Report No. CS/E 92-010, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Engineering.

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.

Jacob, B. L. and Mudge, T. N. (1998). A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 295–306, San Jose, CA.

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.

Kandiraju, G. B. and Sivasubramaniam, A. (2002). Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proc. Int. Conf. on Measurement and Modeling of Computer Systems*, pages 129–139, Marina Del Rey, CA.

Kane, G. and Heinrich, J. (1992). *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ.

Kilburn, T., Edwards, D., Lanigan, M., and Sumner, F. (1962). One-level storage system. *IRE Trans. on Electronic Computers*, EC-11(2):223–235.

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.

Krishnan, V. and Torrellas, J. (1999). A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. on Computers*, 48(9):866–880.

Kroft, D. (1981). Lockup-free instruction fetch/prefetch cache organisation. In *Proc. 8th Int. Symp. on Computer Architecture (ISCA '81)*, pages 81–84.

Lee, D., Baer, J.-L., Calder, B., and Grunwald, D. (1995). Instruction cache fetch policies for speculative execution. In *Proc. 22nd annual Int. Symp. on Computer Architecture*, pages 357–367, S. Margherita Ligure, Italy.

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.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. (1992). The Stanford DASH multiprocessor. *Computer*, 25(3):63–79.

Lin, W.-F., Reinhardt, S., and Burger, D. (2001). Reducing DRAM latencies with an integrated memory hierarchy design. In *Proc. 7th Int. Symp. on High Performance Computer Architecture (HPCA-7)*, pages 301–312, Monterrey, Mexico.

Lo, J., Emer, J., Levy, H., Stamm, R., and Tullsen, D. (1997). Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. on Computer Systems*, 15(3):322–354.

Lo, J. L., Barroso, L. A., Eggers, S. J., Gharachorloo, K., Levy, H. M., and Parekh, S. S. (1998). An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. 25th Int. Symp. on Computer Architecture (ISCA '98)*, pages 39–50, Barcelona, Spain.

Machanick, P. (1996). *An Object-Oriented Library for Shared-Memory Parallel Simulations*. PhD Thesis, Department of Computer Science, University of Cape Town.

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

Machanick, P., Salverda, P., and Pompe, L. (1998). Hardware-software trade-offs in a Direct Rambus implementation of the RAMpage memory hierarchy. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 105–114, San Jose, CA.

Maekawa, M., Oldehoeft, A., and Oldehoeft, R. (1987). *Operating Systems: Advanced Concepts*. Benjamin/Cummings, Menlo Park, CA.

Martel, I., Ortega, D., Ayguadé, E., and Valero, M. (1999). Increasing effective IPC by exploiting distant parallelism. In *Proc. 13th Int. Conf. on Supercomputing*, pages 348–355.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117.

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.

Nakatani, T. and Ebcioğlu, K. (1990). Using a lookahead window in a compaction-based parallelizing compiler. In *Proc. 23rd Annual Workshop and Symp. on Microprogramming and Microarchitecture*, pages 57–68, Orlando, FL.

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.

Patterson, D. and Hennessy, J. (1994). *Computer Organisation and Design: The Hardware/Software Interface*. Morgan Kauffmann, San Francisco, CA.

Postiff, M. A., Green, D. A., Tyson, G. S., and Mudge, T. N. (1999). Limits of instruction level parallelism in SPEC95 applications. *Computer Architecture News*, 27(1):31–34. presented at INTERACT-3 Workshop on Interaction between Compilers and Computer Architectures, part of ASPLOS VIII, San Jose, CA, October 1998.

Rivers, J. A., Tyson, G. S., Davidson, E. S., and Austin, T. M. (1997). On high-bandwidth data cache design for multi-issue processors. In *Int. Symp. on Microarchitecture*, pages 46–56, Research Triangle Park, NC.

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.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2002). *Operating System Concepts*. John Wiley, New York, 6th edition.

Skadron, K., Ahuja, P. S., Martonosi, M., and Clark, D. W. (1999). Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Trans. on Computers*, 48(11):1260–1281.

Smith, A. (1982). Cache memories. *ACM Computing Surveys*, 14(3):473–530.

Topham, N., González, A., and González, J. (1997). The design and performance of a conflict-avoiding cache. In *Proc. 30th Int. Symp. on Microarchitecture (MICRO-30)*, pages 71–80, Research Triangle Park, NC.

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.

Waldspurger, C. and Weihl, W. (1993). Register relocation: flexible contexts for multithreading. In *Proc. 20th Annual Int. Symp. on Computer Architecture (ISCA '93)*, pages 120–130, San Diego, CA.

Wall, D. (1991). Limits of instruction level parallelism. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-4)*, pages 176–188, Santa Clara, CA.

Wheeler, B. and Bershad, B. (1992). Consistency management for virtually indexed caches. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pages 124–136.

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

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

Yeh, T.-Y. and Patt, Y. N. (1993). A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. 20th Annual Int. Symp. on Computer Architecture*, pages 257–266, San Diego, CA.

Young, C. and Smith, M. D. (1999). Static correlated branch prediction. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 21(5):1028–1075.